

Coding Guidelines and Undecidability

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—The C and C++ programming languages are widely used for the implementation of software in critical systems. They are complex languages with subtle features and peculiarities that might baffle even the more expert programmers. Hence, the general prescription of *language subsetting*, which occurs in most functional safety standards and amounts to only using a “safer” subset of the language, is particularly applicable to them. Coding guidelines are the preferred way of expressing language subsets. Some guidelines are formulated in terms of the programming language and its implementation only: in this case they are amenable to automatic checking. However, due to fundamental limitations of computing, some guidelines are *undecidable*, that is, they are based on program properties that no current and future algorithm can capture in all cases. The most mature and widespread coding standards, the MISRA ones, explicitly tag guidelines with *undecidable* or *decidable*. It turns out that this information is not of secondary nature and must be taken into account for a full understanding of what the guideline is asking for. As a matter of fact, undecidability is a common source of confusion affecting many users of coding standards and of the associated checking tools. In this paper, we recall the notions of decidability and undecidability in terms that are understandable to any C/C++ programmer. The paper includes a systematic study of all the undecidable MISRA C:2012 guidelines, discussing the reasons for the undecidability and its consequences. We pay particular attention to undecidable guidelines that have decidable approximations whose enforcement would not overly constrain the source code. We also discuss some coding guidelines for which compliance is hard, if not impossible, to prove, even beyond the issue of decidability. Findings and lessons learned are reported along with some concrete suggestions to improve the state of the art.

I. INTRODUCTION

Coding guidelines are restrictions in the way high-level programming languages can be used to construct programs. The role played by such guidelines in ensuring system safety and security is steadily increasing in importance due to the following factors:

- the increased criticality of the software-controlled functions in modern systems;
- the sheer complexity and number of traps and pitfalls in the most commonly used programming languages, such as C and C++;

- the consequent ease with which programming errors are committed.

Due to this, *language subsetting*, i.e., the prescription to only use a restricted subset of the language such that the potential of committing possibly dangerous mistakes is reduced, is mandated or strongly recommended by the most important functional safety standards.¹ Language subsetting is generally implemented by the enforcement of coding guidelines.

An important distinction among coding guidelines concerns those that are only defined in terms of the actual source code and of the toolchain used to translate it to executable code,² and those making reference to other information, such as requirements, specifications and designs. The former would be amenable to full automatic checking if it was not for *undecidability*: this is a fundamental limitation of any sufficiently-expressive programming language whereby there is no general mechanical procedure that can decide whether or not a program has certain properties. *Undecidable properties* are those that cannot be decided by a general mechanical procedure, i.e., they cannot be implemented by an algorithm.

Among the undecidable properties are *all* the more interesting ones, those that every software engineer would want to decide: the presence or absence of buffer or numeric overflows, invalid pointer dereferences, divisions by zero, ..., they are all undecidable. As a result, guidelines that only depend on the source code and on the language implementation are further subdivided into *decidable guidelines*, those that, at least in principle, can be verified automatically, and *undecidable guidelines*, those that cannot.

It is not difficult to give a rule of thumb for recognizing decidable guidelines: they only depend on program properties that are known at compile time, like

- the types of the objects;
- the names and the scopes of identifiers;
- syntactic properties of the source code, like the presence of *goto*'s.

¹Such as IEC 61508 [1] (industrial, generic), ISO 26262 [2] (automotive), CENELEC EN 50128 [3] (railways), RTCA DO-178C [4] (aerospace) and FDA's *General Principles of Software Validation* [5] (medical devices).

²For example, C99 has 112 implementation defined behaviors [6] and C18 has 119 [7]. These influence so many aspects of source code interpretation that we can say C source code cannot be assigned any meaning unless full details are available on the implementation-defined behaviors of the used translation toolchain.

* 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.

On the other hand, a guideline is almost certainly undecidable if it depends on conditions that are only known at run-time, like

- the values contained in modifiable objects;
- whether control reaches particular points.

Decidability has deep consequences on automatic analysis techniques, particularly on static analysis. For decidable guidelines, it is theoretically possible (i.e., modulo the availability of sufficient computational resources) for a tool to emit a message if and only if the rule is violated. In contrast, for undecidable guidelines, any tool will have to implement an approximated decision algorithm, that is, one that only in some cases can provide a *yes/no* answer and in the remaining cases gives a *don't know* answer. Not all tools have implementations matching this level of sophistication (which is indeed challenging for reasons that go beyond the scope of this paper). In fact, several tools only provide *yes/no* answers implying that, in reality, they are unable to recognize the *don't know* cases. So what they do is either:

- they keep silent in cases where there can actually be a violation: these have false negatives and are unsuitable to safety- or security-related development;
- they emit violation messages in cases where there may not be a violation: these have false positives but no false negatives, so they can be used for safety- and security-related development even though, if there are too many false positives, effectiveness of the tool is low;
- a combination of the above, for tools having both false negatives and false positives.

Due also to these aspects, tool users are often confused about undecidable guidelines and how tools report their possible violation.

In this paper, we study the relationship between undecidability and coding guidelines in widely-used coding standards. We focus particularly on MISRA C, which is the most authoritative and most widespread subset for the C programming language [8]. MISRA C, whose first edition was published in 1998 [9] and directed to the automotive industry, has become a *de facto* standard for the development of high-integrity and high-reliability systems in all industry sectors. The intended audience for this paper is constituted by:

- users of coding standards and of tools supporting them;
- organizations defining coding standards;
- producers of tools supporting coding standards.

The paper is structured as follows: Section II introduces undecidable program properties in a way that is accessible to every software developer; Section III illustrates undecidable MISRA C:2012 guidelines and classifies them according to their nature and the techniques with which they can or cannot be checked; Section IV focuses on guidelines concerning unreachable and “dead” code, which have peculiarities distinguishing them from other undecidable guidelines; Section V discusses the findings of this research work, makes some concrete proposals for improvement, and draws some conclusions

on the desirability of undecidable guidelines vs decidable ones; Section VI wraps up.

II. UNDECIDABLE PROGRAM PROPERTIES

A function is said to be *computable* if there exists an algorithm that, given enough resources, will always produce the correct output for each given input.³ For simplicity, let us consider functions with one input, a natural number, and a Boolean output where we use 1 as the representation of *true* and 0 as the representation of *false*. The function $e: \mathbb{N} \rightarrow \mathbb{N}$ given by

$$e(x) = \begin{cases} 1, & \text{if } x \text{ is even,} \\ 0, & \text{if } x \text{ is odd,} \end{cases}$$

is clearly computable. It is important to observe that the condition for computability is the existence of the algorithm, independently from the fact that we have the algorithm and we know how to implement it. Consider the following functions:

$$f(x) = \begin{cases} 1, & \text{if exactly } x \text{ consecutive '5's appear} \\ & \text{in the decimal expansion of } \pi, \\ 0, & \text{otherwise;} \end{cases}$$

$$g(x) = \begin{cases} 1, & \text{if at least } x \text{ consecutive '5's appear} \\ & \text{in the decimal expansion of } \pi, \\ 0, & \text{otherwise.} \end{cases}$$

Function $g: \mathbb{N} \rightarrow \mathbb{N}$ is computable: it is either the function that always gives 1 (if the decimal expansion of π contains sequences of consecutive ‘5’s of any length), i.e.,

$$g(x) = 1, \tag{1}$$

or there exists $k \in \mathbb{N}$ such that

$$g(x) = \begin{cases} 1, & \text{if } x \leq k, \\ 0, & \text{if } x > k. \end{cases} \tag{2}$$

In the first case an algorithm computing g is the following, where `natural` is a type that encodes natural numbers:

```
natural g(natural x) {
    return 1;
}
```

In the latter case an algorithm computing g has the form

```
natural g(natural x) {
    return (x <= k) ? 1 : 0;
}
```

³This section contains a drastic simplification of a small part of what is presented in any standard university course on Turing-computability. All the mentioned results are well known since the early 1950’s and the main ideas were established in the 1930’s thanks to the work of Gödel, Church, Péter, Turing, Kleene and Post [10].

The fact that we do not know yet⁴ which is the right algorithm, i.e., whether there exists k such that (2) holds or, instead, whether (1) holds, does not really matter: g is computable.

The same thing cannot be said for function f : its shape may be so complex, jumping back and forth from 0 to 1 in a way that no algorithm can capture, or maybe the jump pattern is expressible by an algorithm: we simply do not know yet.⁴

Let us consider a generic programming language and let $\mathbb{P}$ be the infinite set of all its programs. Let also $\mathbb{I}$ be the infinite set of all inputs for the programs in $\mathbb{P}$. For a program $P \in \mathbb{P}$ and a possible input $I \in \mathbb{I}$, a *program property* is a statement of the form “program P [predicate] when run with input I .” Examples of predicates are:

- 1) has 3 if-then-else’s;
- 2) terminates;
- 3) divides by zero;
- 4) does not terminate.
- 5) does not divide by zero.

For each $P \in \mathbb{P}$ and each $I \in \mathbb{I}$ let $p(P, I)$ mean “ P has property p when run on input I .” Let us call $p_1, \dots, p_5$ the properties corresponding to examples 1–5 in the enumeration above, e.g., p_2 is “ P terminates when run on input I .”

For a property p , let us consider the decision function for p , which we will denote by ϕ_p : it takes a program P (any one in $\mathbb{P}$), an input I (any one in $\mathbb{I}$), and responds with 1 if program P has property p when run on I ; it responds with 0 otherwise. More formally $\phi_p: \mathbb{P} \times \mathbb{I} \rightarrow \mathbb{N}$ is defined, for each $P \in \mathbb{P}$ and each $I \in \mathbb{I}$, by

$$\phi_p(P, I) = \begin{cases} 1, & \text{if } p(P, I) \text{ holds,} \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

We say that p is decidable if and only if ϕ_p is computable. Observe that p_1 is clearly decidable: an implementation of ϕ_{p_1} will disregard input I completely, and inspect P to count the if-then-else’s, returning 1 if P has exactly 3 of them and 0 otherwise.

What about p_2 ? Program termination is notoriously undecidable for all sufficiently expressive programming languages, such as C, Pascal, Python, and all general-purpose languages: these languages are called *Turing-equivalent* and have the property that, if a function is computable at all, then it is computable by a program written in any of those languages.

To see that p_2 is undecidable, consider the following argumentation in a subset of the C programming language where we fixed *all* the implementation-defined behaviors. Let $\mathbb{C}$ be the subset of C programs where:

- 1) we systematically avoid all unspecified behaviors by the use of temporary variables and sequence points;
- 2) we systematically define, in an arbitrary way, all undefined behaviors.

For example, to fix the implementation-defined behaviors, let us say something like “we stick to the dialect of C18

⁴Unless a new result in number theory has been published after this paper was written.

```
#include <stdio.h>
/* Returns 1 if P points to the name of a file
   containing a valid restricted C program source
   that terminates when called with the input
   contained in the file whose name is pointed
   to by I; returns 0 otherwise.
   The function works perfectly for every
   combination of its inputs and always returns
   the correct result in finite time. */
int halts(const char *const P,
          const char *const I) {
    /* ... */
}

int main(int argc, char **argv) {
    const char * const P = argv[1];
    while (halts(P, P)) {
        printf("%s will terminate on %s\n", P, P);
    }
    printf("%s will not terminate on %s\n", P, P);
    return 0;
}
```

Fig. 1. Source file `halts.c`, with the impossible-to-write function `halts()` and a main program exercising it

implemented by GCC version v for target z with a fixed set of options including `-std=c18` and `-pedantic-errors`. For an example of point 1, we never write something like $z = f() + g()$; instead, we write, e.g.,

```
{
    T1 x = f();
    T2 y = g();
    z = x + y;
}
```

where T1 and T2 are the return types of `f()` and `g()`, respectively. For an example of point 2, we never use integer division directly, as in $z = x / y$; instead, we write, e.g.,

```
int intdiv(int a, int b) {
    return (b == 0) ? 0 : (a / b);
}
...
z = intdiv(x, y);
```

and we always use `intdiv()` when dividing two quantities of (promoted) type `int`. Note that we are in no way restricting the expressive power of the language: a program relying on unspecified or undefined behavior is ill-formed anyway [7].

Now suppose, towards a contradiction, that we can actually implement ϕ_{p_2} , that is, we can implement in $\mathbb{C}$ a function that, given any program P and any input I , will give 1 if P terminates on I and will give 0 if P does not terminate on I . This amounts to say that we are able to complete the body of function `halts()` in Figure 1 so as to implement its specification. However, if `halts()` can be written in $\mathbb{C}$ then it can also be called in $\mathbb{C}$ in the way indicated in the same figure. Then we can compile the program and execute it as follows:

```
$ gcc -std=c18 ... halt.c -o halt.exe
```

```

#include <stdio.h>
/* Returns 1 if P points to the name of a file
   containing a valid restricted C program source
   that performs a division by zero when called
   with the input contained in the file whose name
   is pointed to by I; returns 0 otherwise.
   The function works perfectly for every
   combination of its inputs and always returns
   the correct result in finite time. */
extern int divbyzero(const char *const P,
                    const char *const I);

int halts(const char *const P,
          const char *const I) {
    /* Modify program in P by (1) replacing all
       divisions by zero with equivalent code, and
       (2) adding a statement performing a division
       by zero just before any of its explicit or
       implicit return points; write the resulting
       program in a file whose name is contained
       in P1. */
    const char *P1 = NULL;
    /* ... */
    return divbyzero(P1, I);
}

```

Fig. 2. Deriving a decision procedure for termination from a decision procedure for division by zero

```
$ halt.exe halt.c
```

For the indicated execution of `halt.exe` there are only two possibilities:

- 1) The program prints


```
halt.c will terminate on halt.c
halt.c will terminate on halt.c
... [infinite repetitions]
```

 so that in fact *it will not terminate!*
- 2) The program prints


```
halt.c will not terminate on halt.c
```

 but *it has just terminated!*

We reached a contradiction in both cases, meaning that the function `halts()` cannot be written. Note that it is not just a matter that we do not know how to write it, we simply cannot: we will *never* be able to write it.

It can be proved that *all* non-trivial semantic properties of programs are undecidable: these are all properties that depend on the contents of writable memory locations and/or on the fact that a certain program point can be reached or not. Take division by zero, for instance: if we were able to implement ϕ_{p_3} , say, with a C function `divbyzero()`, then we would be able to implement `halts()` as shown in Figure 2. As `halts()` cannot be implemented, also `divbyzero()` cannot. The same holds for all non-trivial program properties: absence of buffer overflows and any other run-time error, reachability of program points and so on. The proofs of these can all be obtained as variations of the argument presented here: let the bad thing happen if and only if the program terminates on the given input. In this sense termination can be called “the father of all undecidable

problems.” However, termination is *semidecidable*, meaning that the following function is computable:

$$\psi_{p_2}(P, I) \begin{cases} = 1, & \text{if } P \text{ terminates in input } I, \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

The algorithm is simple: run P on input I ; if and when termination takes place, return 1; otherwise keep executing P on I . Undecidability of termination implies that, in general, we cannot do better than that. But consider the property p_4 of *non-termination*: this is not even semidecidable: in general, at no point in time are we allowed to conclude that, having not observed termination, termination cannot take place later. The same holds for the property p_5 of *non-division-by-zero*. Worse than that, as the program input cannot typically be known in advance, the properties we are interested in are the *universal* ones, that is, instead of ϕ_p of (3), we would need

$$v_p(P) = \begin{cases} 1, & \text{if } p(P, I) \text{ holds for each } I \in \mathbb{I}, \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Of course, if ϕ_p is not computable, v_p is also not computable. Thus, if a property is undecidable, its universal counterpart is also undecidable. Take termination: its universal counterpart is called *universal termination* and a program *universally terminates* if it terminates for each input. Universal termination not only is undecidable, but would remain undecidable even if we had an *oracle* for ordinary termination, that is, if we had some sort of magic behaving like ϕ_{p_2} : this would not help as there typically are infinitely many inputs for a program.

Finally, the fact that universal termination is undecidable, allows us to easily prove that program equivalence is also undecidable. Figure 3 shows how a decision procedure for program equivalence could be turned into a decision procedure for universal termination.

III. MISRA RULES AND UNDECIDABILITY

MISRA rules are explicitly classified as *decidable* or *undecidable* according to whether answering the question “Does this program comply?” can be done algorithmically. All the considerations of the previous section apply, taking into account that all MISRA guidelines are based on *universal program properties*.

Out of the 199 guidelines in MISRA C:2012 Revision 1 with Amendments 2 and 3 [8], [11], [12], 181 are *rules*, that is, guidelines such that information concerning compliance is fully contained in the source code and in the implementation-defined aspects of the used C language implementation. Of the 181 rules, 40 are undecidable, of which 12 are *mandatory*, 23 are *required*, and 5 are *advisory*.⁵ Table I provides a synopsis of such undecidable rules. It contains one row for each rule, whose identifier is written in boldface if mandatory, in *italics* if advisory, or in normal font if required. For each

⁵MISRA-compliant code must follow *mandatory* guidelines: deviation is not permitted. MISRA-compliant code shall follow every *required* guideline: a *formal deviation* is needed where this is not the case. *Advisory* guidelines are recommendations that should be followed as far as is reasonably practical. See MISRA Compliance:2020 [13] for details.

TABLE I
SYNOPSIS OF THE UNDECIDABLE RULES IN MISRA C:2012

	flow undecid.	numeric undecid.	pointee undecid.	side eff. undecid.	flow ins. approx.	type approx.	other approx.	coverage	not provable	definition issues
<i>Rule 1.2</i>										x
Rule 1.3	x	x	x	x						
Rule 1.5	x	x								
Rule 2.1	x							x		
Rule 2.2	x								x	x
<i>Rule 8.13</i>	x					ooo				
Rule 9.1	x		x				o			
Rule 12.2	x	x								
Rule 13.1	x			x	o		ooo			
Rule 13.2	x		x	x			o			x
Rule 13.5	x			x	o		ooo			
Rule 14.1	x	x					o			
Rule 14.2	x	x	x				o			
Rule 14.3	x							x		
Rule 17.2	x		x			o				
<i>Rule 17.5</i>	x		x							
<i>Rule 17.8</i>	x		x			ooo				
Rule 17.9	x				ooo					
<i>Rule 17.11</i>	x				oo					
Rule 18.1	x	x	x							
Rule 18.2	x		x		o					
Rule 18.3	x		x		o					
Rule 18.6	x		x				o			
Rule 19.1	x		x		o					
Rule 21.13	x	x				oo				
Rule 21.14	x		x			oo				
Rule 21.17	x	x	x							
Rule 21.18	x	x	x							
Rule 21.19	x		x			oo				
Rule 21.20	x		x							
Rule 22.1	x		x				o			
Rule 22.2	x		x				o			
Rule 22.3	x									x
Rule 22.4	x		x			oo				
Rule 22.5	x				oo					
Rule 22.6	x						o			
Rule 22.7	x					oo				
Rule 22.8	x						oo			
Rule 22.9	x						oo			
Rule 22.10	x						oo			

rule, information is summarized in the following columns:

flow undecid. for rules whose undecidability directly depends on the tracking of control-flow;

numeric undecid. for rules whose undecidability directly depends on the tracking of numeric values;

pointee undecid. for rules whose undecidability directly depends on the tracking of pointee addresses;

side effects undecid. for rules whose undecidability directly depends on the tracking of side effects;

flow ins. approx. for rules admitting useful *flow insensitive*, *sound* approximations;

type-based approx. for rules admitting *type-based*, *sound* and *decidable* approximations;

other approx. for rules admitting *decidable* approximations using other techniques;

coverage for rules whose compliance can only be checked by dynamic analysis and tracking of test coverage;

not provable for rules such that compliance is generally not

provable, either programmatically or manually;

definition issues for rules whose definition has issues discussed in this paper.

In the columns labelled as approximations, the approximability is classified using notations ‘o’, ‘oo’ and ‘ooo’. The number of ‘o’s in the notation indicates how easy it would be for the developers to satisfy the extra requirements due to the approximation. That is:

- A single ‘o’ indicates an approximation having occasional violations whose avoidance is troublesome and/or seriously limiting the developers. Deviation might be appropriate.
- A ‘oo’ indicates an approximation such that avoiding violations would need some care and possibly some rewriting, but that leads to better and provably correct code. Deviation might be considered as an alternative to refactoring.
- A ‘ooo’ indicates an approximation where deviating

```

#include <stdio.h>
/* Returns 1 if P1 and P2 point to the name of
files containing two valid restricted C program
sources that have the very same behavior for
each input.
The function works perfectly for every
combination of its inputs and always returns
the correct result in finite time. */
extern int equiv(const char *const P1,
                const char *const P2);

int always_halts(const char *const P) {
    /* Modify program in P by adding a "return 1"
statement just before any of its explicit or
implicit return points; write the resulting
program in a file whose name is contained
in P1. */
    const char *P1 = NULL;
    /* ... */
    /* Set P2 to the name of a file containing
the program "int main() { return 1; }". */
    const char *P2 = NULL;
    /* ... */
    return equiv(P1, P2);
}

```

Fig. 3. Deriving a decision procedure for universal termination from a decision procedure for program equivalence

violations is never recommendable and fixing the code is straightforward.

Consider the following MISRA C:2012 rule:

Rule 22.5: A pointer to a FILE object shall not be dereferenced

The reason why this rule is *flow undecidable* is its *flow sensitivity*, meaning that the rule is not violated if a pointer to a FILE object is dereferenced in unreachable code:

```

FILE *p;
/* ... */
if (always_false_in_this_configuration(/* ... */) {
    FILE f = *p; // Unreachable: not a violation.
}

```

The obvious *flow-insensitive*, decidable approximation consists in flagging all dereferences of pointers to FILE objects independently from reachability. In this and other cases, *flow sensitivity* (ubiquitous in undecidable rules as you can see in Table I) does more harm than good: in the example above, the reason that **Rule 22.5** is not violated is because the *then* branch is unreachable. What is the point of this exemption? In fact the unreachable code is in violation of Rule 2.1 (discussed in Section IV).

For another example, consider the following guideline:

Rule 17.8: A function parameter should not be modified

This is undecidable for two reasons. The first is, again, *flow sensitivity*: a modification happening on a branch that is not reached is not a violation. Consider the following

Example 1:

```

void f(uint32_t x) {
    if (x < 0) { // If always x >= 0 on entry...
        x = 0;   // ... Rule 17.8 is not violated.
    }
    /* ... */
}

```

Flow sensitivity is not the only cause of undecidability for Rule 17.8. Consider the following

Example 2:

```

extern void g(uint32_t *p);

void f(uint32_t x) {
    /* ... */
    g(&x); // Rule 17.8 violation?
    /* ... */
}

```

Knowing whether *Rule 17.8* is violated depends on knowing whether function *g()* modifies the pointee of the argument it received on input, which also depends on the tracking of pointee addresses, and this is undecidable. In this and other cases, the authors of this paper believe that the latitude allowed by undecidability has insufficient rationale. This is why, for several rules, Table I goes beyond crossing *flow insensitive approximation* by crossing *type approximation*. We call “type approximation” one that can be expressed in a stricter type system than the standard C type system.⁶ Note that static type approximations are, by definition, flow insensitive and decidable.⁷ Concerning *Rule 17.8*, a sensible type approximation would go along the lines “Function parameters should be considered as read-only.” A static analyzer can clearly check this, in particular by checking that no explicit and implicit casts can circumvent the writing prohibition, and flag the violation in Example 2.

We will now go through other rules in Table I, explaining their classification and their relationship with undecidability.

Rule 1.2: Language extensions should not be used.

The rationale of *Rule 1.2* is that programs relying on extensions will be difficult to port to a different language implementation. In addition, extensions make *compiler qualification*, as mandated by functional safety standards, more expensive, as it requires in-house development of test cases for the extensions [16].

An interesting thing about *Rule 1.2* is that it is the only MISRA C:2012 rule that is tagged both as *Undecidable* and *Single Translation Unit*, meaning that all violations involve

⁶This notion is not new to MISRA C practitioners: MISRA C:2012 *essential type model* along with the guidelines that are based on it, define a type system which is stronger than the C type system.

⁷*Static typing* is in contrast with *flow-sensitive typing*, where the type of expressions may depends on their position in the control flow. In flow-sensitive type systems, the type of an expression may be updated to a more specific type following an operation validating the subtype. For example, just after `p = malloc(sizeof(int))` the type of `p` may be an encoding of “null pointer or pointer to the beginning of a block in the heap”, but within the *then* branch of a subsequent if-then-else guarded by `p != NULL`, the type of `p` may be updated to “pointer to the beginning of a block in the heap.” The Rust language and the C-rusted dialect of C, which we will mention later in the paper, are based on flow-sensitive typing [14], [15].

a single translation unit only, i.e., the compiler and not the linker. However, the point with language extensions is that, by definition, they are not known in advance. Consider the following C fragment:

```
extern void f(char *p);
void g(void);

void g(void) {
    char a[9] = {};
    return f(a);
}
```

This contains an empty initializer and returning of a `void` expression, which are undefined in all versions of the C standard. Nonetheless the GCC documentation does not document them as extensions.⁸ Can an undocumented compiler feature be accepted as a legitimate language extension? Probably not: as the presence of documentation is crucial and checking that is a human activity, *Rule 1.2* should probably be a directive.

Rule 1.3: There shall be no occurrence of undefined or critical unspecified behaviour

Rule 1.3 covers all undefined and critical unspecified behaviors that are not covered by other rules: there are many of them, so the crosses given in the corresponding row in Table I represents a summary.

Rule 8.13: A pointer should point to a `const`-qualified type whenever possible

The reason why *Rule 8.13* is undecidable is that the missing `const` qualification might impact on code that is unreachable. As unreachable code is flagged by Rule 2.1 (see Section IV), the rationale for accepting undecidability is weak. We believe the stronger, decidable version, would be more useful without constraining programmers too much; hence in Table I, in the column labeled *type-based approx.*, the rule has ‘ooo’.

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

Missing initialization of automatic variables is the origin of many defects and vulnerabilities. **Rule 9.1** is undecidable because it rules out reading uninitialized stack cells: capturing this with high precision in static analysis is challenging, especially when arrays are involved. The rule, however, has sensible decidable approximations. The simplest one is to “always initialize automatic variables at declaration time.” This is less extreme than it might seem: MISRA C:2012 itself hints at this direction when defining Rule 2.2 (which will be examined in Section IV), by specifying that initializations may be kept even when redundant. Most importantly, wholesale initialization of automatic variables is now optionally implemented by major compilers (GCC from version 12, Clang from version 8), with at most negligible slowdowns and speedups

⁸See <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>, last accessed and checked on October 6, 2022.

in some cases [17], [18], [19].⁹ It is interesting to note how, in front of an important rule that is targeted by basically all bug finders (with false negatives and/or false positives due to the rule undecidability), a very speed-sensitive community like the one revolving around the Linux kernel is seriously considering the systematic use of such options.

Rule 14.1: A *loop counter* shall not have *essentially floating type*

This rule, together with Rule 14.2, aims at introducing in C a sort of *determinate iteration* construct like Pascal’s `FOR` looping construct.¹⁰ This ideal, which is not completely achieved, rests on restrictions that are applied to C’s `for` loops. As the restrictions are semantic in nature, both Rule 14.1 and Rule 14.2 are undecidable. The type restriction given in the headline of Rule 14.1 might be surprising, as static type restrictions are decidable. The undecidability is due to the specification of *loop counter*: a loop counter is variable that satisfies three conditions, two of which are undecidable [8, Section 8.14]. Syntactic, fully decidable restrictions are of course possible and would have the advantage of fully achieving the goal of having determinate iteration in C.

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

There are two reasons why this rule is undecidable: flow sensitivity and function pointers. A variant that is flow insensitive and forbids the use of function pointers would be decidable. When function pointers are needed to implement, e.g., callbacks, a type-based approximation (effectively limiting the use of function pointers so that recursive calls via them are impossible) would be decidable and would cover most cases that occur in embedded system programming.

Rule 21.13: Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value `EOF`

Undecidability of this rule comes, besides flow-sensitivity, from the undecidability of value tracking. A type-based approximation, where the static analyzer enforces constraints on a type only capable of representing the value `EOF` and those representable by an `unsigned char`, provides a decidable way of ensuring compliance.

Rule 21.14: The Standard Library function `memcmp` shall not be used to compare null terminated strings

This is undecidable for the same reasons as **Rule 21.13** and the same discussion applies. Indeed, null-terminated strings deserve not just a fictitious type that only manifests itself within the static analyzer: they deserve an explicit `typedef` name so that developers are very aware when they manipulate null-terminated strings and not ordinary character arrays.

⁹The reason for the occasional speedups seems to be that systematic zero-initialization improves superscalar execution in the CPU due to the breaking of dependencies.

¹⁰A *determinate iteration* construct is one such that the maximum number of iterations is known before the first iteration begins. None of the looping constructs of C have this property.

Rule 21.19: The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or, `strerror` shall only be used as if they have pointer to const-qualified type

The very same approach described for *Rule 17.8* can be used.¹¹

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

For this rule as well as **Rule 22.2** it is possible to use an *ownership* model similar the one implemented in Rust [15] and based on *flow-sensitive typing*. This is what is done in C-rusted [14].

Rule 22.3: The same file shall not be open for read and write access at the same time on different streams

The problem with this rule is that it cannot be checked automatically well beyond the problem of undecidability. The notion of being “the same file” is not definable at the C source level even taking into account the implementation-defined behaviors of the implementation. Such notion depends on peculiarities of file systems and on their current state: relative paths, hard links, symbolic links, logical drives and other file system features are such that the check for compliance requires a lot of information that only knowledgeable humans can provide. In other words, this MISRA guideline should probably be a directive.

Rule 22.7: The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`

This rule is undecidable because of flow sensitivity and because of the semantic notion of *unmodified value*. A type-based approximation drastically restricting what can be done on the return values of the indicated functions would be decidable.

Rule 22.8: The value of `errno` shall be set to zero prior to a call to an *errno-setting-function*

This rule, as for the associated Rule 22.9 and Rule 22.10, is undecidable because it is overly permissive in the forms and positioning of the operations that zero and test `errno`. Syntactic restrictions would result into decidable guidelines without constraining the programmer in an unacceptable way.

IV. GUIDELINES ON UNREACHABLE AND DEAD CODE

MISRA C:2012 has two required rules to deal with useless, and thereby possibly undesirable, code. The first one is:

Rule 2.1: A project shall not contain *unreachable code*

There, *unreachable code* is code that cannot be executed unless the program has undefined behavior. The rationale of the rule is that the presence of unreachable code may:

¹¹This rule might be extended to also cover `strchr()`, `memchr()` and similar functions whose indiscriminate use can circumvent the `const` promise of their parameter.

- Indicate an error in the program’s logic: as the compiler is allowed to remove unreachable code (but it is not required to do so),¹² why is the code there?
- Waste resources and prevent optimizations in the case the compiler and the linker do not remove the unreachable code.

To this, we can add that unreachable code that ends up in the executable program is, per se, a security issue: an attacker might exploit another vulnerability in order to actually reach that code and achieve its malicious goals.

Compliance with Rule 2.1 cannot be proved by means of static analysis alone.¹³ Static analysis can reveal that some functions are never called because, e.g., they never occur in explicit function calls and their addresses are never taken; the user can supplement this information by annotating functions that are only called via assembly code or as interrupt services routines. Static analysis can also reveal that a certain condition is always true or always false, when this does not depend on external inputs, or that an expression used to control a `switch` statement can only take certain values. But, in general, making sure all code is actually reachable requires dynamic analysis: 100% statement coverage has to be reached for that purpose, with a test-suite that is a *subset of the input space for the program*. Note that we are not talking of *unit testing* here (reaching 100% statement coverage via unit tests does not ensure compliance), but rather of *system testing*.

Rule 2.2: There shall be no *dead code*

MISRA C:2012 Rule 2.2 is a required rule that defines *dead code* as “Any operation that is executed but whose removal would not affect program behaviour,” further specifying that “unreachable code is not dead code as it cannot be executed.”¹⁴ Three citations from functional safety standards are provided: IEC 61508-7 Section C.5.10 [20], DO-178C Section 6.4.4.3.c [4], and ISO 26262-6 Section 9.4.5 [21]. The first two references are also cited for Rule 2.1, and in fact:

IEC 61508-7 Section C.5.10 This section does not define *dead code*, but gives a terse introduction to data-flow analysis. It gives three examples of use, two of which do overlap with the MISRA C:2012 definition of *dead code*: these concern values being unnecessarily written to memory, a.k.a. *dead stores* in other literature.

DO-178C Section 6.4.4.3.c Section 6.4.4.3 is titled “Structural Coverage Analysis Resolution”, which already hints at the fact that MISRA C:2012 definition of dead code is not compatible with DO-178C definition. Such definition

¹²Even though not explicitly mentioned, the linker (and in some cases *only* the linker), can also remove unreachable code from the executable, while not being required to do so.

¹³Experts might object that *symbolic model checking* may be sufficient in some cases. However, since this is based on the synthesis of test cases that are validated by *symbolic* or *concolic* (concrete mixed with *symbolic*) execution, we assimilate this technique to dynamic analysis.

¹⁴The notion of *operation* is not explicitly defined in MISRA C:2012 or in the C Standard. We interpret *operation* as to signify “anything that in the C Standard is called *operation*: integer operations, floating-point operations, arithmetic operations, bitwise operations, atomic operations, synchronization operations, pointer operations, ...”.

can be found in Annex B, Glossary: “Dead code — Executable Object Code (or data) which exists as a result of a software development error but cannot be executed (code) or used (data) in any operational configuration of the target computer environment. It is not traceable to a system or software requirement.” DO-178C also defines a different notion of “deactivated code.”

ISO 26262-6 Section 9.4.5 This section of ISO 26262-6 is concerned with structural coverage. To exemplify the use of structural coverage methods, two examples are given:

“EXAMPLE 1 Analysis of structural coverage can reveal shortcomings in requirements-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality.”

“EXAMPLE 2 A rationale can be given for the level of coverage achieved based on accepted dead code (e.g. code for debugging) or code segments depending on different software configurations; or code not covered can be verified using complementary methods (e.g. inspections).”

So, while ISO 26262 does not give a definition of *dead code*, it clearly states that dead code can be found by analyzing structural coverage.

Summarizing, the definition of *dead code* given in MISRA C:2012 is not compatible with the definitions given in ISO 26262 and DO-178C. The only overlap between the functional safety standards cited by MISRA C:2012 and its notion of *dead code* is given by *dead stores*. In other words, the named standards define *dead code* as *code that cannot be executed plus data that is written and cannot be read*. This is a much narrower definition than the MISRA C:2012 definition as “operations that can be removed without affecting program behaviour.”

MISRA C++:2008 has the following required

Rule 0-1-9 There shall be no dead code.

This is different from MISRA C:2012 Rule 2.2 in that *dead code* is characterized as “any executed statement whose removal would not affect program output [...]”. In the sequel, in order to avoid confusion with the definitions used in functional safety standards, we will use the expression *effectless code* instead of the MISRA C:2012 and the MISRA C++:2008 notions of *dead code*.

In the online version of the *SEI CERT C Coding Standard*¹⁵ this topic is covered by the following *Recommendation*:

MSC12-C Detect and remove code that has no effect or is never executed

SEI CERT C Coding Standard uses the word *Recommendation* as opposed to *Rule*: whereas rules are normative (though not necessarily amenable to automatic analysis), “recommendations are meant to provide guidance that, when followed, should improve the safety, reliability, and security of software

systems.” Following MISRA terminology, CERT-C *Recommendation MSC12-C* would be an advisory directive, whereas MISRA C:2012 Rule 2.2 is a required rule.

A wholesale ban on effectless code, like the one required by MISRA C:2012 Rule 2.2, discourages sane things like the ones shown in Figure 4.

Even a wholesale ban on dead stores is undesirable. Consider the following example

```
void transmit_octet(const uint8_t octet) {
    uint8_t mask = 1U;
    for (uint8_t bit = 0; bit < 8; ++bit) {
        transmit_bit(octet & mask);
        mask <<= 1U;
    }
}
```

The shift-assignment to *mask* on the last iteration is a dead store. However, modifying the source code in order to avoid it would almost certainly decrease code quality for no gain, not even on efficiency. The point is that the rationale against dead stores is quite weak: while it is true that they may indicate a programming mistake, often they do not, and compilers are quite good at detecting them and optimizing them out when there is incentive to do so.

An important issue with MISRA C:2012 Rule 2.2 is that it is the only undecidable MISRA C:2012 rule that is also unprovable. There is neither a static or dynamic analysis, nor a sensible review process that can decide whether an operation can be removed from a program without affecting its behavior.

Note the difference with respect to Rule 2.1: achieving 100% statement coverage proves that a project is compliant. Moreover, for MISRA C:2012

Rule 14.3: Controlling expressions shall not be invariant

achieving 100% branch coverage proves compliance. Of course, static analysis techniques can pinpoint some instances of non-compliance with these rules: what remains to be proved for compliance can be done via dynamic analysis. In contrast, for Rule 2.2, proving compliance is generally impossible: it is not just that program behavior equivalence is strongly undecidable, as we saw in Section II. In order to prove compliance with Rule 2.2, one should, for each combination of operations in the program (the number of which is exponential in the total number of operations), prove that removing that combination preserves program behavior. In other words, proving compliance with respect to Rule 2.2 would require answering with *no* an exponentially large number of questions (exponential in the size of the program) of the form “is the transformed program, where we have deleted some operations, behaviorally equivalent to the original program?” Answering these question programmatically cannot be done, it cannot be done via dynamic analysis, and it cannot be done manually as program equivalence requires the same behavior for each of the possible inputs, of which there typically is an infinite number.

¹⁵<https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>, last accessed on October 4, 2022. Note that *Recommendation MSC12-C* is not contained in the printed version [22].

```

x + OFFSET;      // Addition is justifiable, even when OFFSET is defined to be 0.
x * SCALE;       // Multiplication is justifiable, even when SCALE is defined to be 1.
x * sizeof(T);   // Multiplication is justifiable, no matter what the value of
                // sizeof(T) is.
do_X_if_necessary(); // The function may be inline and its body may be empty
                // (e.g., #ifdef-ed out) in this configuration.

typedef enum {
    BIT0 = 1U << 0, // Justifiable (no-op) shift by 0 positions.
    BIT1 = 1U << 1,
    BIT2 = 1U << 2
} Bit_Masks;

```

Fig. 4. Examples of undesirable violations of MISRA C:2012 Rule 2.2

Guidelines for which compliance is practically impossible to be proved serve no purpose. For instance, when confronted with rule MISRA C:2012 Rule 2.2, users will either:

- 1) unknowingly fake compliance (possibly with the complicity of tool vendors making claims that they cannot actually make about the coverage of the rule); or,
- 2) when they are knowledgeable enough, they will raise a project deviation saying that they did their best and that they have confidence that remaining effectless code, if any, is not causing problems.

V. DISCUSSION

In this section we briefly discuss the tradeoff between decidable and undecidable guidelines and we put forward a concrete proposal about the treatment of effectless code.

A. How Good Are Undecidable Coding Guidelines?

Are undecidable coding guidelines good or bad? The attentive reader already knows the position of the authors:

- an undecidable guideline is good when there are no decidable sound approximations for it (that is, one that catches all its violations plus more) or when such decidable approximations would tie the hands of programmers in a way they cannot easily achieve their objectives;
- in all other cases, undecidable guidelines are bad.

Undecidable guidelines are troublesome because they let programmers deal with false positives and/or false negatives: if there are no false negatives then there may be many false positives and these are time consuming to deal with; if there are no or few false positives, then there may be false negatives, in which case developers will have to look for different or additional solutions. Note that this is a universal constant: if a guideline is undecidable, any fully automatic checker will have false positives or false negatives or both.

It is very instructive to observe that, independently from the number of companies and organizations that offer verification tools basically for free to the Linux community:

- 1) for the undecidable undefined behavior caused by reading uninitialized automatic variables, as we have already observed, they are looking at compiler options to solve the problem;

- 2) for the various undecidable undefined behaviors related to memory management errors, they are looking at Rust [23], [24], [25], [26], [27], [28], [29], [30].

While point 1 involves no effort on the part of the developers, point 2 and the possible move to Rust require a lot of discipline on the part of the programmers, which shows that the Linux community, when it comes to important safety and security matters, is in line with MISRA Compliance:2020, which states that simply satisfying the immediate convenience of the developer does not constitute an acceptable rationale for deviating guidelines [13, Section 4.4].

Note that we are not necessarily proposing to change the rules in Table I so as to make all those with ‘oo’ or ‘ooo’ in at least one *approx.* column decidable. Another possibility is for tools to implement some sound decidable approximations for those rules

B. An Alternative To the Strict Ban on Effectless Code

As we have seen in Section IV, in general there is no way to actually, fully comply with MISRA C:2012 Rule 2.2 or CERT-C *Recommendation MSC12-C*. For projects with MISRA-compliance requirements this is a problem: as they cannot claim compliance with MISRA C:2012 Rule 2.2 or MISRA C++:2008 Rule 0-1-9, they will have to deviate. For the relatively few cases that a tool can detect:

- the code can be amended if this has a positive effect on code quality;
- a deviation has to be raised otherwise, as recommended by MISRA Compliance:2020 [13] (code quality always comes first).

In any case, a deviation will have to be raised *in addition* to all that, simply because nobody can know whether there are other undetected violations of the rule. Unfortunately, this does not match any of the allowed rationales for deviation allowed by MISRA Compliance:2020 (code quality, access to hardware, integration or use of suitably qualified adopted code). Pragmatically, projects will have no choice other than raising a project deviation with a justification along with the following lines:

Peer review gives us confidence that no evidence of errors in the program’s logic has been missed due to

undetected violations of Rule 2.2, if any. Testing on time behavior gives us confidence on the fact that, should the program contain dead code that is not removed by the compiler, the resulting slowdown is negligible.

A possible solution to rectify the situation is by replacing MISRA C:2012 Rule 2.2 with a directive whose headline can be something like

Unjustified effectless code shall be minimized.

Note the similarity with required directive

Dir 4.1: Run-time failures shall be minimized

Dir 4.1 takes a very pragmatic approach to a much more serious problem than effectless code: this is reasonable as ensuring the absence of run-time errors is impossible as is ensuring the absence of effectless code.

The notion of *unjustified* can be defined as follows: code is *unjustified* if it has both of the following attributes:

- 1) it does not help understanding of the algorithm;
- 2) it does not come from the natural abstraction of the algorithm so that it can be applied in different situations (e.g., on different architectures and/or on different configurations).

In particular effectless code is justified if it arises from an abstraction process. That can be:

- data abstraction: macro expansions, macro definitions;
- control abstraction: loops, recursion.

Figure 5 provides examples of compliance and non-compliance to this hypothetical directive.

Compliance to the directive requires planning and documenting activities, possibly involving static analysis and dynamic analysis techniques, along the lines of Dir 4.1 (i.e., with reference to design standards, test plans, static analysis configuration files and code review checklists).

VI. CONCLUSION

Undecidability is an inescapable limitation of computing whereby only purely-syntactic properties of programs written in languages such as C and C++ are algorithmically verifiable or refutable: all other properties are undecidable, meaning that there will never be a general algorithm that can decide whether a program has or does not have the property. As a result, most of the program properties associated to program safety and security requirements are undecidable.

Coding guidelines that embody the language-subsetting requirement of many functional safety standards are constrained between two conflicting goals:

- 1) they have to prevent bad things from happening;
- 2) they have to be acceptable to developers.

Goal 2 implies the coding guideline should be directly targeted at preventing the bad thing. Given that the possibility of occurrence of the bad thing is usually undecidable, the conjunction of goal 1 with goal 2 tends to favor undecidable guidelines. The tradeoff changes between communities and with time:

- developers of critical software in highly-regulated industry sectors are more willing to exchange a little bit of inconvenience with the strong guarantees that decidable guidelines can provide;
- even highly-unregulated communities, like the one revolving around the Linux kernel, seem now inclined to accept more restrictions: something that, only a few years ago, would have been vehemently rejected.

While undecidable coding guidelines cannot pragmatically be dispensed with, at least for languages like C and C++, analysis tools can, in several cases, be based on some decidable approximations. Nonetheless, undecidable program properties tend to confuse developers, no matter whether the guidelines used to determine that the program has the property are undecidable or decidable. Many developers are distracted by (sometimes false) thoughts like “There is no problem in my program, why this violation?” without considering that no perfect solution exists and that, by necessity, we need to compromise.

We believe this paper, in which we studied the role played by undecidability in the most widely used coding standard, MISRA C, will help developers in better appreciating what the problems are and which tradeoffs have to be faced. Some of the findings of this research were totally unexpected at the outset: as a result, this work goes beyond its original survey/educational goals by uncovering some real problems and corresponding possible solutions.

Acknowledgments

The MISRA C Guideline headlines are reproduced with permission of *The MISRA Consortium Limited*. We are grateful to Federico Serafini (University of Parma and BUGSENG) and Nicola Vetrini (BUGSENG) for their comments on draft versions of this paper.

REFERENCES

- [1] IEC, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. Geneva, Switzerland: IEC, Apr. 2010.
- [2] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.
- [3] CENELEC, *EN 50128:2011: Railway applications — Communication, signalling and processing systems - Software for railway control and protection systems*. CENELEC, Jun. 2011.
- [4] RTCA, *SC-205, DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Dec. 2011.
- [5] *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>.
- [6] ISO/IEC, *ISO/IEC 9899:1999/Cor 3:2007: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2007, Technical Corrigendum 3.
- [7] —, *ISO/IEC 9899:2018: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2018.
- [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 software*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jul. 1998.

```

x + 0;           // Unjustified addition.
x + OFFSET;     // Justified addition, even when OFFSET is defined to be 0.

x * 1;          // Unjustified multiplication.
x * SCALE;      // Justified multiplication, even when SCALE is defined to be 1.
x * sizeof(T);  // Justified multiplication, no matter what the value of sizeof(T) is.

do_X_if_necessary(); // Justified function call, unless it can be argued that for all present and future
                    // project configurations, the function has no influence on the program behavior.

for (i = 0; i < NUM_REPETITIONS; ++i) { // Entire loop justified, even if NUM_REPETITIONS
    do_things();                       // expands to 0 or 1 in this configuration.
}

// Saturate.
x = (x > MAX) ? MAX : x; // Justified, even if X is never greater than MAX in this configuration.

```

Fig. 5. Examples of justified and unjustified effectless code

- [10] R. I. Soare, *Turing Computability — Theory and Applications*, ser. Theory and Applications of Computability. Springer, 2016.
- [11] MISRA, *MISRA C:2012 Amendment 2 — Updates for ISO/IEC 9899:2011 Core functionality*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.
- [12] —, *MISRA C:2012 Amendment 3 — Updates for ISO/IEC 9899:2011/2018 Phase 2 — New C11/C18 features*. Norwich, Norfolk NR3 1RU, UK: The MISRA Consortium Limited, Oct. 2022.
- [13] —, *MISRA Compliance:2020 — Achieving compliance with MISRA Coding Guidelines*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.
- [14] R. Bagnara, A. Bagnara, and F. Serafini, “C-rusted: The advantages of Rust, in C, without the disadvantages,” 2023. [Online]. Available: <https://arxiv.org/abs/2302.05331>
- [15] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, online version available at <https://doc.rust-lang.org/book/> and maintained at <https://github.com/rust-lang/book>, last accessed on February 22, 2022.
- [16] R. Bagnara, A. Bagnara, and P. M. Hill, “A rationale-based classification of MISRA C guidelines,” in *embedded world Conference 2022 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2022, pp. 440–451.
- [17] JF Bastien. (2019, Oct.) Mitigating undefined behavior: Security mitigations through automatic variable initialization. LLVM Foundation. Presentation at the *2019 LLVM Developers’ Meeting — Bay Area*, video last accessed on October 7, 2022. [Online]. Available: <https://www.youtube.com/watch?v=I-XUHPimq3o>
- [18] K. Cook and Q. Zhao. (2021) Compiler features for kernel security. Presentation at the *Linux Plumbers Conference 2021*, slides last accessed on October 7, 2022. [Online]. Available: <https://lpc.events/event/11/contributions/1026/>
- [19] Q. Zhao. (2021) Security improvements in GCC. Presentation at the *Linux Plumbers Conference 2021*, slides last accessed on October 7, 2022. [Online]. Available: <https://lpc.events/event/11/contributions/1001/>
- [20] IEC, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems — Part 7: Overview of Techniques and Measures*. Geneva, Switzerland: IEC, Apr. 2010.
- [21] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 6: Product development at the software level*. Geneva, Switzerland: ISO, Dec. 2018.
- [22] CERT, *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Software Engineering, Carnegie Mellon University, 2016, 2016 edition.
- [23] J. Salter, “Linus Torvalds weighs in on Rust language in the Linux kernel,” *Ars Technica*, Mar. 2021. [Online]. Available: <https://arstechnica.com/gadgets/2021/03/linus-torvalds-weighs-in-on-rust-language-in-the-linux-kernel/>
- [24] B. Cantrill, “Is it time to rewrite the operating system in Rust?” *InfoQ*, Jan. 2019. [Online]. Available: <https://www.infoq.com/presentations/os-rust/>
- [25] J. Wallen, “Let the Linux kernel Rust,” *TechRepublic*, Jul. 2021. [Online]. Available: <https://www.techrepublic.com/article/let-the-linux-kernel-rust/>
- [26] S. J. Vaughan-Nichols, “Linus Torvalds on where Rust will fit into Linux,” *ZDNet*, Mar. 2021. [Online]. Available: <https://www.zdnet.com/article/linus-torvalds-on-where-rust-will-fit-into-linux/>
- [27] —, “Where rust fits into linux,” *The Register*, Nov. 2021. [Online]. Available: https://www.theregister.com/2021/11/10/where_rust_fits_into_linux/
- [28] L. Tung, “Google backs effort to bring Rust to the Linux kernel,” *ZDNet*, Apr. 2021. [Online]. Available: <https://www.zdnet.com/article/google-backs-effort-to-bring-rust-to-the-linux-kernel/>
- [29] M. Melanson, “Rust in the Linux kernel: ‘good enough’,” *The New Stack*, Dec. 2021. [Online]. Available: <https://thenewstack.io/rust-in-the-linux-kernel-good-enough/>
- [30] N. Elhage, “Supporting Linux kernel development in Rust,” *LWN.net*, Aug. 2020. [Online]. Available: <https://lwn.net/Articles/829858/>