# Jumpstarting MISRA compliance via the integration of static analysis into Open Source CI systems: best practices and key elements from TrustedFirmware.org



## Executive Summary

## Introduction

TrustedFirmware.org is well aware of the challenges involved in meeting the needs of the users and regulatory agencies and how complicated it has become to support leading edge secure technologies that are required to meet the requirements of modern systems. Examples can be found in the evolution of the Arm architecture, with the Armv9 revision introducing various security enhancements around memory safety and confidential computing.

The endeavour to properly develop, integrate and validate the most privileged system software in a way that is ready to pass compliance requirements is not for the lighthearted. But, thanks to TrustedFirmware.org, reference implementations are available. These implementations are integrated and validated on "real" hardware platforms. These permissively-licensed Open Source reference implementations provide a valuable starting point for solution and platform providers alike. Providers can leverage these implementations vs attempting to develop such a complex secure software stack from scratch, thus allowing providers to spend more of their resources on feature differentiation.

With it becoming apparent that TrustedFirmware.org is primarily deployed in Mobile/Cloud/Embedded-IoT segments, there is also the need in providing a valuable jumpstart for Platform/Solution providers who are leveraging the code in industrial, medical, automotive, and other solutions requiring platform level regulatory compliance. An example in ensuring TrustedFirmware projects provide the most value as a reference platform in such

environments has been the inclusion of MISRA compliance testing. This led TrustedFirmware.org to the partnership with BUGSENG and the integration of ECLAIR, BUGSENG's high-quality compliance tool.

## C is a powerful language to be handled with care

The C programming language has been in use for half a century and remains one of the most utilised programming languages. The reasons behind this longevity are compelling: C allows the generation of fast compiled code for any architecture, it is standardised by ISO, there is a wide availability of tools and libraries and a long history of application, including in safety-critical industries. Yet, programming in unrestricted C is known to be dangerous. The main reasons are:

1. The language has many aspects that are not fully defined. In turn this is the downside of C's important advantages:
    a. speed of compiled code is obtained at the expense of directly mapping high-level constructs to machine instructions that have different semantics on different architectures (e.g., this is why shifting signed integers is not fully defined);
    b. speed of compiled code is also a consequence of the lack of run-time error checking;
    c. ISO standardisation and wide availability of compilers from different vendors implies different compilers implement some parts of the language differently from one another.
2. Terseness of the language, which is a good thing, has the downside that C is easily abused and misunderstood. Implicit conversions are governed by quite intricate rules, which many programmers do not know in enough detail. The wide supply of predefined operators, with precedence and associativity rules that may surprise developers, and of control-flow constructs quite often result in programs that are difficult to understand and, thus, opaque to peer review.

In the course of 2022, around 250,000 CVEs (security vulnerabilities) have been filed, and around half of those concern C code. This state of affairs is pushing regulators to adopt stricter rules with the objective of making safety and security of software-based systems a priority (see, e.g., Stop Passing the Buck on Cybersecurity: Why Companies Must Build Safety Into Tech Products).

While these call-to-actions are driven by the need of facing rampant cybersecurity threats, safety is also a primary concern. Functional safety standards in all industry sectors, such as IEC 61508 (general), ISO 26262 (automotive), CENELEC EN 50128 (railways), DO-178C (aerospace), IEC 60880 (nuclear power), and IEC 62304 (medical devices) have been developed to tackle the issue of programming language issues for decades: the pragmatic solution identified by the industry is called *language subsetting*: in essence, when programming critical systems, only the safer subsets of programming languages are used. This applies to all languages, not just C. When it comes to C, MISRA C is the most authoritative C subset for the development of safety- and security-critical software.

# MISRA C

1973 is the year when the essential elements of what is today called C were completed; the compiler was also strong enough for Dennis Ritchie and Ken Thompson to rewrite the Unix kernel for the PDP-11 in C during the summer of that year.
So C is 50 years old, but MISRA C, whose first edition is dated July 1998, is 25 years old. As C evolved and matured along with ISO standardisation (C90, C99, C11, C18 and, soon, C2x), MISRA C evolved in order to follow the language evolution and to strengthen the guarantees provided to the producers of critical software (MISRA C:1998 was followed by MISRA C:2004, MISRA C:2012 and MISRA C:2023, along with several addenda, amendments and corrigenda that were published over the years).
MISRA C defines a subset of the C programming language that greatly reduces or mitigates the possibility of committing programming mistakes that may impact the safety and security of devices containing software (i.e., nowadays, all devices!).
The MISRA C subset of C is defined by means of coding *guidelines*. In turn, guidelines are divided into *directives* and *rules*: for the latter, compliance depends only on the code and on the way the particular compiler used (along with the options with which it is used) resolves the many implementation-defined aspects of the language. So, if all the code is available (but often, libraries are not provided in source format), if all the code is written in C (but for many projects some code is written in assembly) and if it was not for undecidable program properties, MISRA C rules could be checked in a fully automatic way by means of static analysis tools. Differently from *rules*, compliance with the MISRA C *directives* does not depend only on the code and on the compilation process. Design, specification and even the programmer's intentions and knowledge have to be taken into account. Thus, in the case of directives, tools may assist in checking compliance if they are provided with the extra information. MISRA C Directive 1.1 is a good example which states

> *Dir 1.1 Any implementation-defined behaviour on which the output of the program depends shall be documented and understood*

C18 (ISO/IEC 9899:2018) has 119 implementation-defined behaviours: they cover the size and alignment of data objects, whether the plain `char` type is signed or unsigned, the details of the representation of and access to bitfields, and lots of other things. Most compilers allow controlling some implementation-defined behaviours by means of compilation options. It is clearly impossible to reliably program in C without an understanding of how the used compiler with the used compilation options resolves the implementation-defined behaviours. While a static analysis tool cannot of course judge the adequacy of the documentation, let alone its understanding on the part of the involved personnel, a tool like ECLAIR (see the dedicated section later in the paper) can automatically detect which such behaviours may influence the output of the program. For typical embedded system software, they are on the order of 30 to 40, which implies the remaining 80 to 90 need not be documented.
A very important MISRA C rule prescribes the use of standard C:

> *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*

At first sight, this rule might seem surprising: isn't it the task of the compiler to make sure text that is not C is rejected? In fact, compilers cannot be trusted: they often do accept constructs not defined by the language. For instance, GCC accepts the following code:

```
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 published versions of the C standard. Nonetheless the GCC documentation does not document them as extensions, so they cannot be accepted as such and *Rule 1.1* is violated here. Another important aspect of *Rule 1.1* concerns the fact that a conforming compiler does not need to generate a diagnostic when a translation limit (such as the maximum number of cases in a `switch` statement, or the maximum level of file inclusion nesting) is exceeded, and an executable may be generated that does not work as expected.
Note that language extensions documented as such and supported by the compiler are allowed by MISRA C, even though advisory Rule 1.2 suggests treating them with care: language features that are outside the supported versions of C have not been considered when developing the MISRA guidelines; moreover, extensions will pose problems when compiler qualification is required (who will provide a sufficient set of test cases for the extensions?).

Another interesting MISRA C guideline is

> *Rule 3.2 Line-splicing shall not be used in // comments*

Line-splicing is when a back-slash at the end of a line causes two physical source lines of code to be merged into one logical source line of code. If a `//` commented line ends with a back-slash followed by a new-line, then the following line is part of the comment, and, if this was not intended, an important line of code may be lost. The following example shows how a path separator at the end of the comment may accidentally comment out the next line of code:

```
// see critical.* in c:\project\src\
critical_function();
```

Mandatory MISRA C *Rule 9.1* says we must ensure that stack variables always have a value when they are read:

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

Whereas objects with static storage duration are automatically initialised to zero unless initialised explicitly, objects with automatic storage duration (i.e., those allocated on the stack) are not automatically initialised and can therefore have indeterminate values: attempts of reading indeterminate values are *undefined behaviour*. In the following example, if function `f()` is ever called with a negative actual parameter, the behaviour is undefined, which means the program can just do anything:

```
int32_t f(int32_t a) {
  int32_t x ;
  if (a >= 0) {
      x = 2;
  }
  return x;
}
```

Note that *undefined behaviour* is a technical term coming from the C Standard: "behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements," where "no requirements" means absolutely no requirements: crashing, erratic behaviour of any kind, formatting the hard disk, … a standard-conforming compiler can produce code that really can do anything. Normally, it means the compiler assumes undefined behaviour does not happen; if it does happen, the programmer has violated the contract with the language and any warranty is void. In other words, a program that has undefined behaviour is totally unpredictable.
Another class of circumstances where the meaning of C programs is not fully defined is called *unspecified behaviour*: this is when there are two or more options to implement one language aspect and the C Standard gives the compiler freedom to make a choice in each individual instance (which implies that compiler does not have to document the choice and does not have to ensure any sort of consistency when the choice has to be made repeatedly). For example, the order in which subexpressions are evaluated and the order in which side effects take place is unspecified for most operators. For instance, the following program might print '`Hello world! Hello world! `' or '`world! Hello world! Hello `' or '`world! Hello Hello world! `' or '`Hello world! world! Hello `':

```
#include <stdio.h>

int hello(void) {
  return printf ("Hello ");
}

int world(void) {
  return printf ("world! ");
}
```

```
int main () {
  int x = hello() + world()
  int y = hello() + world();
  return x + y;
}
```

Compliance with the following MISRA C guideline avoids this issue:

> *Rule 13.2 The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders*

The rule also prevents some instances where lack of sequentialization leads to undefined behaviour:

- modifying an object more than once between two sequence points (for instance, in the expression `++y + ++y`);
- modifying and reading an object between two sequence points, unless reading is necessary to store in the object (for instance, in the expression `++y + y`).

The order in which actual parameter expressions are evaluated in function calls is also unspecified. As reading `volatile` variables is a persistent side effect, the following program has unspecified behaviour and violates *Rule 13.2*:

```
extern volatile uint16_t TCNT1;
extern volatile uint16_t TCNT2;
extern int32_t f(uint16_t tc1, uint16_t tc2);

int main (void) {
  return f(TCNT1 , TCNT2);
}
```

The problem can be easily avoided by the introduction of temporary variables and leveraging the fact that statement-terminating semicolons are sequence points:

```
extern volatile uint16_t TCNT1;
extern volatile uint16_t TCNT2;
extern int32_t f(uint16_t tc1, uint16_t tc2);

int main (void) {
  uint16_t tc1 = TCNT1;
  uint16_t tc2 = TCNT2;
  return f(tc1, tc2);
}
```

The selection of guidelines that has just been presented is representative of the spirit of MISRA C: this has to do with error prevention more than bug finding. While violations of

certain guidelines (such as *Rule 9.1*) clearly represent programming mistakes, in the case of other guidelines violations may or may not be defects depending on the programmer's intentions. In all cases, violations are an invitation to reflect on things like: “Did we really intend to do this?”, or “Is this thing I wrote and the way I wrote it understandable by my colleagues or by myself one month from now?”, or “What if this function is called with an argument such-and-such?” This is why the *deviation process* (whereby justifiable non-compliances can be authorised and recorded as *deviations*) is an essential part of MISRA C: the point of a MISRA C guideline is not “You should not do that”' but, rather, “This could be dangerous: you may only do that if (1) it is needed, (2) it is safe, and (3) a peer can easily and quickly be convinced of both (1) and (2).” In fact, when a high-quality MISRA-checking tool is available, a useful way of thinking about MISRA C and the processes around it is to consider them as an effective way of conducting a guided peer review to rule out most C language traps and pitfalls.

## TrustedFirmware Open CI and ECLAIR

TrustedFirmware uses a comprehensive Continuous Integration (CI) system “Open CI.” The system is completely public, allowing contributors across different companies and around the world to participate in TrustedFirmware development and testing. The system runs over 3000 builds across several TrustedFirmware Projects (such as TF-A, TF-M, and MBed TLS) daily, and as such is quite busy and thorough. In addition to performing daily builds, OpenCI also runs a number of static and dynamic checks, ranging from simple code convention compliance to code coverage analysis.

One of the features that members of the TrustedFirmware.org project prioritised for inclusion into Open CI over the last 1-2 years has been a comprehensive MISRA checker. In the context of product releases, demonstrating adherence to MISRA standards has become an essential requirement, especially in safety-critical industries like automotive, medical, rail, and aviation, among others. Ensuring demonstrable MISRA compliance entails more than just running an automated compliance checker; it encompasses various aspects such as documentation, training, process planning, configuration & change management, and product testing, among others. This enhances the processes and tooling already in place within Open CI that can contribute to MISRA compliance such as source code version control (git), code reviews (Gerrit), documentation, process management, and testing. It is still ultimately left to the product developer to attain MISRA compliance.

One very important aspect of demonstrating compliance involves utilising MISRA static code analysis tooling on the source codebase. Through a partnership with BUGSENG and the integration of ECLAIR into Open CI, TrustedFirmware now offers support for this capability. Again, it's important to note that the tooling itself doesn't magically make a product fully MISRA-compliant when using TrustedFirmware Open Source software components. However, it provides a strong foundation by conducting baseline MISRA analysis supporting the software developer teams in refactoring TrustedFirmware.org projects to meet the requirements of this baseline MISRA scan. This facility is now provided for both TF-A and TF-M projects and may be extended to other projects supported by TrustedFirmware.org. This new service adds to TrustedFirmware.org projects being a valuable starting point when

MISRA compliance is needed for companies seeking to incorporate TrustedFirmware.org code into their products.

ECLAIR has a number of features which make it an excellent choice for using in a CI system:

- First class-support for using in batch mode (all tools support operations from the command line, comprehensive text-based configuration files, etc.).
- Analysis and report generation are separate stages allowing for flexibility in structuring CI jobs.
- As much as possible information about the source codebase is captured during the analysis phase, while additional filtering can be applied at report generation phase.
- Reports produced are standalone - either plain text, or HTML-based. HTML reports are actually dynamic, offering UI with diagramming, filtering capabilities, source code locations preview, etc. - all this without 3rd-party software, with just a user's web browser.

From the implementation point of view, the Open CI system consists of Jenkins CI server software, with most of the individual CI jobs running in Docker containers. ECLAIR integration into Open CI thus largely involves preparing a suitable container image to run in. One of the challenges was support for ECLAIR licensing model - being proprietary software, ECLAIR requires transient connection to a networked licence server to perform its operations. Fortunately, there are different licensing modes, including accommodating running in parallelized batch-mode CI systems.

There were several steps to perform in order to enable ECLAIR in Open CI. Normally, MISRA analysis is applied for the intended goal of achieving MISRA compliance and certification. The certification is generally applied to a "product" having a particular certification requirement with a well-defined set of components on a codebase of a particular version, with specific configuration settings, and built with a specific compiler for a specific hardware platform. TrustedFirmware, on the other hand, is essentially a flexible, highly-configurable framework, supporting many features and hardware platforms. Out of this framework, a number of specific products can be built by downstream vendors. This is achieved by tuning various parameters mentioned above. Thus, MISRA compliance at the level of the TrustedFirmware upstream project is not the goal, nor is it the aim of the project. Instead, the aims are to:

- establish and maintain a baseline for a specific level of MISRA compliance for the codebase;
- provide means for project maintainers and contributors to assess and maintain this level of compliance;
- encourage contributors to improve MISRA compliance of the existing codebase;
- provide to the downstream vendors guidelines, sample setup, and best practices towards working on MISRA compliance and product certification, if they choose so.

The current compliance goal of the project is to be clear of "mandatory" MISRA issues (note that MISRA categorises its rules as *mandatory*, *required*, or *advisory*). To help assess this overall goal, Open CI provides a baseline daily build of the entire codebase, with suitable reports published. The results can be accessed via the following link: https://ci.trustedfirmware.org/job/tf-a-eclair-daily/ .

In addition to these baseline builds of the entire codebase, Open CI provides differential (or “delta”) reports for specific patches submitted by the contributors for inclusion into the project. This is actually a fairly unique and important feature of Open CI, enabled by ECLAIR, which has the ability to create such a differential report from the analyses of two different revisions of the codebase. Internally what happens is that the codebase is analysed without the patch, then with the patch applied, and then ECLAIR produces a differential report capturing the effect of the patch on MISRA compliance. The usefulness of such a report cannot be overestimated. As the TrustedFirmware codebase is quite big and has a long history, it has many MISRA issues still active (as mentioned above, only mandatory issues were completely addressed so far). When reviewing changes introduced by a patch, seeing these background/residue issues would complicate assessing what effect the patch has on MISRA compliance, and why it is very helpful to see issues only related to the changes in the patch. In support of the goal of not having any mandatory MISRA issues in the codebase, this prevents the projects from introducing them via new patches. But with the delta analysis feature, both contributors and maintainers can also review violation of required/advisory guidelines and avoid introducing them whenever possible, to at least not grow their number. Additionally, developers working on gradually resolving these issues in the existing codebase, have immediate per-patch feedback that their work goes in the right direction. Examples of the delta reports can be seen via this link: https://ci.trustedfirmware.org/job/tf-a-eclair-delta/ . Each delta analysis CI job also posts a summary report back to the patch review system (Gerrit), so that the original submitter and reviewers have easy access to it.

Summing up, the MISRA testing setup has integrated well into the existing TrustedFirmware Open CI infrastructure, providing both a full report for the entire codebase, and a “delta” report for proposed changes. This will allow the project to maintain its existing level of MISRA compliance and gradually improve it over time. This results in a great headstart for downstream vendors who would like to pursue MISRA compliance for their TrustedFirmware-based product.

## About the ECLAIR Software Verification Platform

The *ECLAIR Software Verification Platform*, ECLAIR for short, is a powerful platform for the automatic verification of C and C++ programs by BUGSENG. ECLAIR has a modern design implementing state-of-the-art technology that gives it a significant competitive advantage on a number of accounts. The platform has many applications:

- coding rule validation, starting with the MISRA coding standards of course;
- metrics: the HIS metrics plus many more are supported that allow software quality to be assessed in terms of complexity, testability, readability and maintainability;
- bug finding based on a very fast static analyzer able to detect and report bugs and weaknesses that can lead to crashes, misbehaviors, and security vulnerabilities;
- stylistic guidelines from BARR-C:2018;
- integrated requirements management tool and support for the bidirectional traceability between requirements/specifications, code and test cases;
- automatic verification of architectural constraints at the software level, instrumental in providing evidence of independence/isolation/segregation/freedom from interference, as mandated by many standards of functional safety and cybersecurity;

- dynamic analysis and automatic generation of test cases (not commercialised yet).

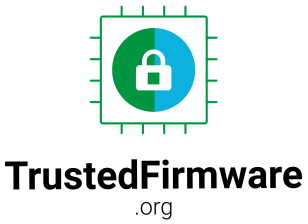
One of the notable features of ECLAIR consists in the fact that the hardest part of the configuration, namely the adaptation to the compiler toolchain and the particular language dialect(s) used in the project, is fully automatic: ECLAIR detects, without user intervention, all the implementation-defined behaviours, including predefined macros, taking into account all options given to the compiler, assembler, linker and librarian. This is particularly suited to open-source projects where code is translated by means of different toolchains and with different options, with both the toolchains and options being updated relatively frequently: with ECLAIR any change in the toolchains or the build procedures is automatically taken into account with total accuracy.
One of the ways in which the design of ECLAIR is crucial in helping developers quickly address its findings concerns the integration with IDEs (all major IDE families are supported: Eclipse, Visual Studio, Visual Studio Code, NetBeans, CLion, …) and with CI/CD systems, including Jenkins, GitLab and GitHub. In particular, with ECLAIR all users have access to fully detailed reports without installing anything; yet, all users have access to private, sophisticated filters (i.e., locally-stored and independent from one another) so that each one of them has the view that is most suitable for the task at hand. Still, using the inexpensive *ECLAIR Client Kit*, users can work within their favourite IDE for extra productivity.

ECLAIR is certified by TÜV SÜD for use in safety-related development according to

- IEC 61508:2010 for any SIL;
- ISO 26262:2018 for any ASIL;
- EN 50128:2011 + A2:2020 for any SIL;
- IEC 62304:2006 + Amd 1:2015 for any software safety class;
- ISO 25119:2018 + Amd 1:2020] for any SRL.

"ECLAIR has provided a very valuable asset to TrustedFirmware.org, its members, and the Community” said Don Harbin, TrustedFirmware.org Community Manager. “The recent release of MISRA testing into our Production CI is the culmination of joint efforts and collaboration between BUGSENG and the Open CI development teams.”

"The integration of ECLAIR into TrustedFirmware's OpenCI constitutes a reference implementation that many organisations can take inspiration from" said Abramo Bagnara, BUGSENG CTO.




## About TrustedFirmware.org

TrustedFirmware.org is an open source project implementing foundational software components for creating secure devices. Trusted Firmware provides a reference implementation of secure software for processors implementing both the A-Profile and M-Profile Arm architecture. It provides SoC developers and OEMs with a reference trusted codebase complying with the relevant Arm specifications. Trusted Firmware code is the preferred implementation of Arm specifications, allowing quick and easy porting to modern SoCs and platforms. This forms the foundations of a Trusted Execution Environment (TEE) on application processors, or the Secure Processing Environment (SPE) of microcontrollers. Visit: https://www.trustedfirmware.org/


## About BUGSENG

BUGSENG is a leading provider of solutions and services for software verification. BUGSENG's ECLAIR Software Verification Platform has been designed to help engineers develop higher-quality software, effectively, by changing the traditional rules of the game. BUGSENG consulting services help industry leaders improving their development processes and complying with functional-safety standards. BUGSENG is also a renowned resource for advanced professional training. Visit: http://BUGSENG.com