TrustedFirmware OpenCI and MISRA testing using ECLAIR

Roberto Bagnara, BUGSENG srl
Paul Sokolovsky, Linaro Developer Services

MISRA C & BUGSENG Tooling Overview

Roberto Bagnara
Software Verification Expert and Evangelist at BUGSENG | Professor of Computer Science | Member of the ISO/IEC JTC1/SC22/WG14 - C Standardization Working Group | Member of the MISRA C Working Group

Linaro

BUGSENG is a leading provider of solutions and services for software verification

Team is composed of highly skilled researchers and software engineers with extensive experience in software verification

Strong presence in the software engineering community:

- MISRA C and C++ Working Groups
- MISRA SQM (Software Quality Metrics)
- ISO/IEC JTC1/SC22/WG14 (C Standardization Working Group)

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

Advantages of the C programming language

There are many strong reasons behind the use of C during the past 50 years:

- C compilers exist for almost **any processor**
- C compiled code is **very efficient** and without hidden costs
- C allows writing compact code (many built-in operators, limited verbosity, ...)
- C is defined by an ISO standard
- C, possibly with extensions, allows easy **access to the hardware**
- C has a long history of usage in critical systems
- C is widely supported by **all sorts of tools**

Disadvantages of C

ISO/IEC JTC1/SC22/WG14, a.k.a. the C *Standardization Working Group*, has always been faithful to the original spirit of the language:

a) Trust the programmer
b) Don't prevent the programmer from doing what needs to be done
c) Keep the language small and simple
d) Provide only one way to do an operation
e) Make it fast, even if it is not guaranteed to be portable
f) Make support for safety and security demonstrable

Point (f) was only added for C11

All the other points are **bad for safety and security**

What is “Behavior”

ISO/IEC 9899:1999 TC3 (N1256) 3.4

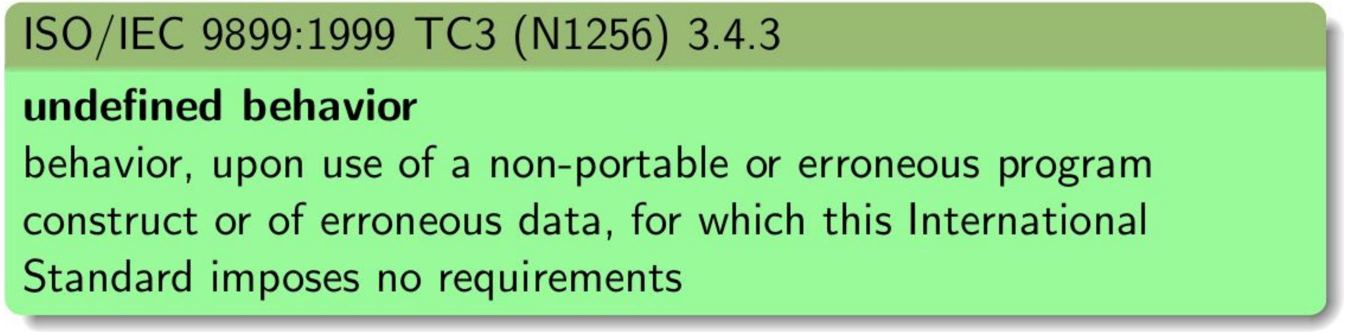
behavior
external appearance or action

As-if rule

The compiler is allowed to do any transformation that ensures that the “observable behavior” of the program is the one described by the standard

True in C, but also in C++, Rust, Go, OCaml, . . .

What is “Undefined Behavior”



ISO/IEC 9899:1999 TC3 (N1256) 3.4.3

undefined behavior

behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

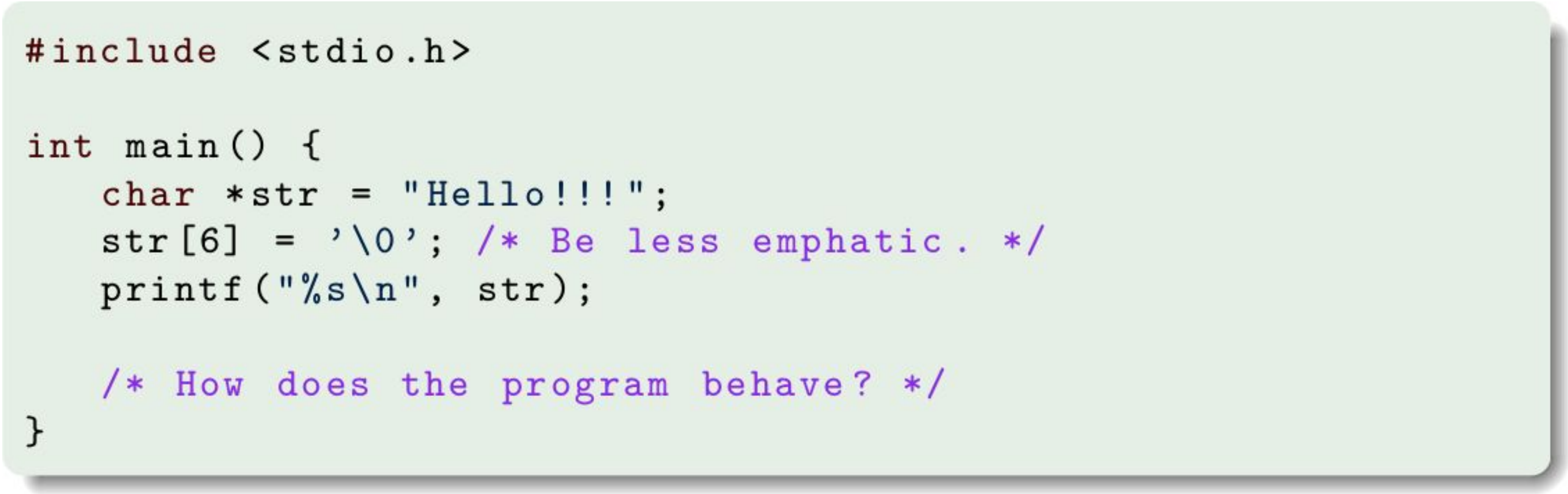
No requirements means **absolutely no requirements**: crashing, erratic behavior of any kind, formatting the hard disk!

Normally it means the compiler assumes undefined behavior **does not happen**

If it does happen, the programmer has violated the contract: **warranty void!**

Undefined Behavior: Examples

- The program attempts to modify a string literal (6.4.5)



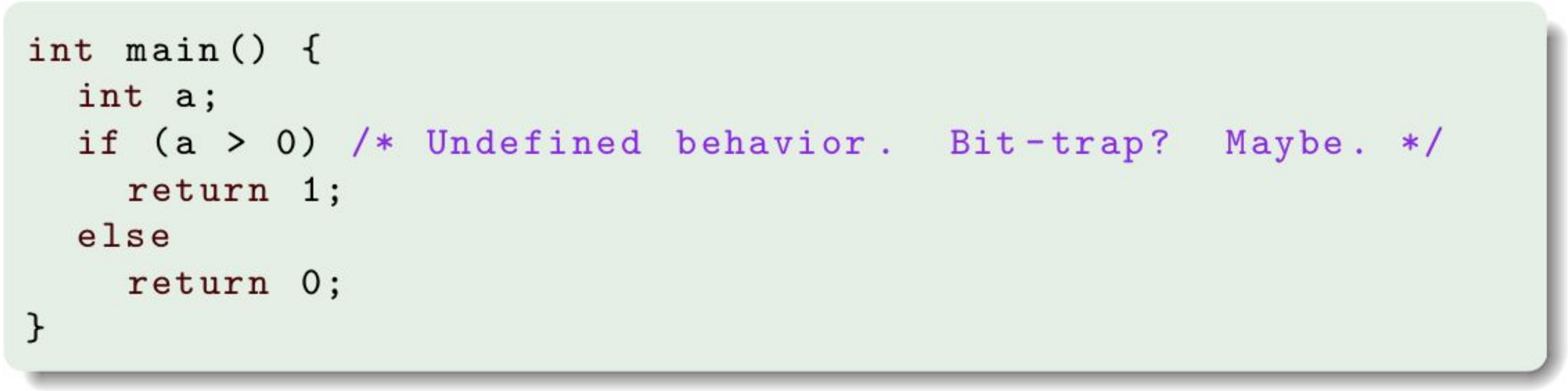
```
#include <stdio.h>

int main() {
    char *str = "Hello!!!";
    str[6] = ’\0’; /* Be less emphatic. */
    printf("%s\n", str);

    /* How does the program behave? */
}
```

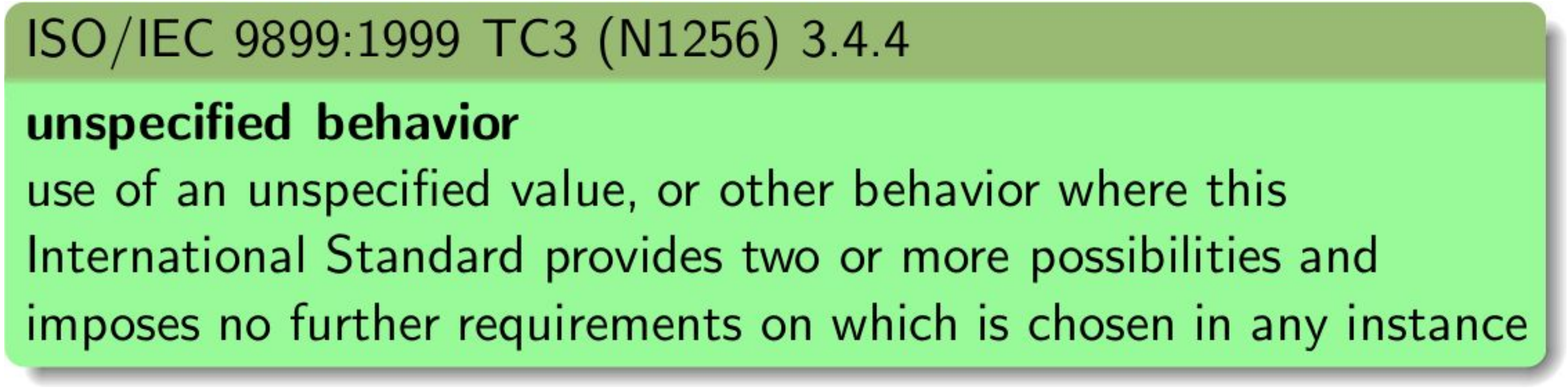
Undefined Behavior: Examples (cont’d)

- The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8)
- A trap representation is read by an lvalue expression that does not have character type (6.2.6.1)



```
int main() {
  int a;
  if (a > 0) /* Undefined behavior.  Bit-trap?  Maybe. */
    return 1;
  else
    return 0;
}
```

What is “Unspecified Behavior”



ISO/IEC 9899:1999 TC3 (N1256) 3.4.4

unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Unspecified Behavior: Example

- The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call (), &&, ||, ?:, and comma operators (6.5)

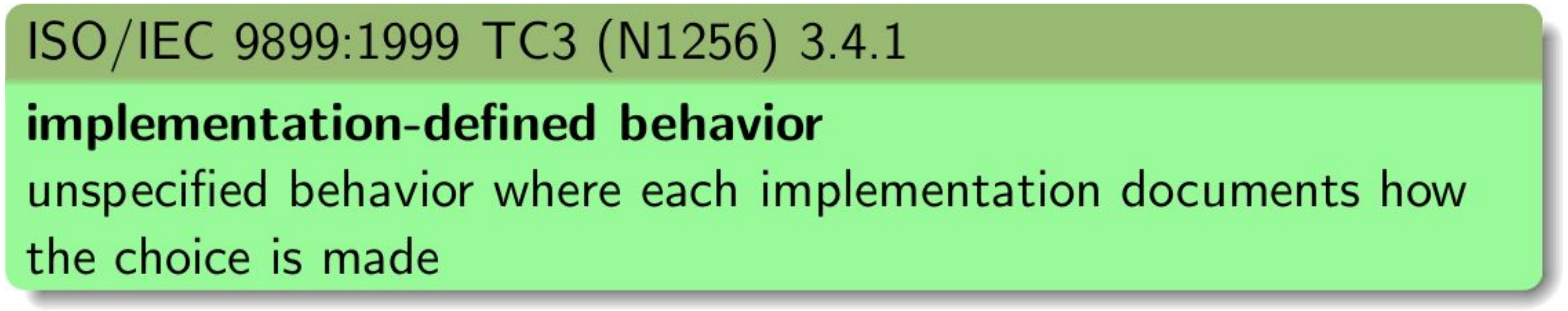
```c
#include <stdio.h>

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

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

int main() {
  return hello() + world();
}
/* Might print: 'Hello world!' or
                'world!Hello '
```

What is “Implementation-Defined Behavior”



ISO/IEC 9899:1999 TC3 (N1256) 3.4.1

implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made

Implementation-Defined Behavior: Example

- Which of signed char or unsigned char has the same range, representation, and behavior as “plain” char (6.2.5, 6.3.1.1)

```
#include <limits.h>
#include <stdio.h>

int main() {
  if (CHAR_MIN == 0)
    fputs("plain char type is unsigned\n", stdout);
  else
    fputs("plain char type is signed\n", stdout);
  /* What is the output? */

  /* Compliant C implementations document this behavior. */
}
```

Why?

We described:

- Undefined behavior
- Unspecified behavior
- Implementation-defined behavior
- (and we glossed over locale-specific behavior)

Why is the standardized language not fully defined?

- Because implementing compilers is **easier**
- Because compilers can generate **faster code**

UB: Signed Integer Overflow

The behavior is undefined when:

An exceptional condition occurs during the evaluation of an expression (6.5)

```
int is_max(int v) {
  return (v + 1 > v) ? 0 : 1;
}
```

Can be compiled as

```
int is_max(int v) {
  return 0;
}
```

UB: Modifying String Literals

The behavior is undefined when:

> *The program attempts to modify a string literal* (6.4.5)

Example: in a program there are literals "Tail" and "HeadTail"

The compiled program can store in memory only "HeadTail" and return the pointer to the fifth character as "Tail"

Changing one string may also change the other, but **the compiler can assume this will never happen**

UB: Shifting Too Much

The behavior is undefined when:

> *An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7)*

```
uint32_t i = 1;
i = i << 32;  /* Undefined behavior. */
```

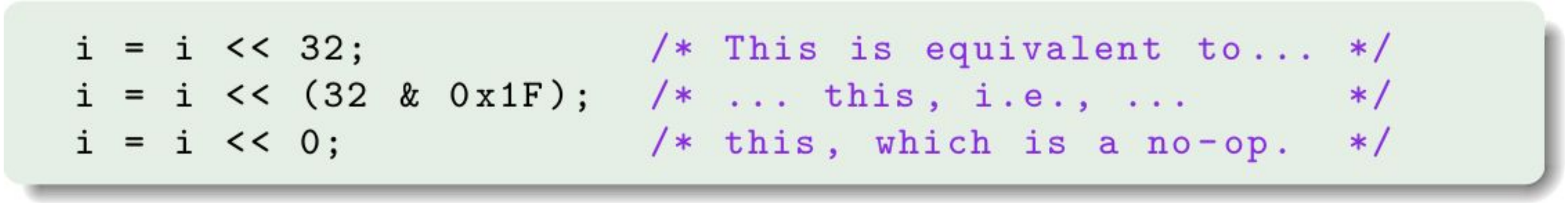
Strange: if I push 32 or more zeros from the right the result should be zero, right?

UB: Shifting Too Much Example

From Intel64 and IA-32 Architectures Manual, page 1706 section "IA-32 Architecture Compatibility":

> *The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.*

Basically, this means that in those machines



```
i = i << 32;            /* This is equivalent to... */
i = i << (32 & 0x1F);   /* ... this, i.e., ...      */
i = i << 0;             /* this, which is a no-op.  */
```

Strength and Weakness of C

The weakness of the C language comes from its strength:

- Ease of writing efficient compilers for almost any architecture ⇒ **non-definite behavior**
- Efficient code with no hidden costs ⇒ **no run-time error checking**
- Many compilers, defined by an ISO standard (must standardize existing practice, many vendors, backward compatibility) ⇒ **non-definite behavior**
- Easy access to the hardware ⇒ **easy to shoot your own foot**
- Compact code ⇒ **the language can be easily misunderstood and misused**

Language Subsetting

Several features of C do **conflict with both safety and security**

For safety-related applications, **language subsetting is crucial**

Mandated or recommended by all safety- and security-related industrial standards:

- CENELEC EN 50128
- IEC 61508
- ISO 26262
- RTCA DO-178C

The **most authoritative** language subset for the C programming language is **MISRA C**

Presentation of the MISRA C Guidelines

Rule 5.6 A *typedef* name shall be a unique identifier

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

On the left, a unique identifier for the guideline composed by a classification as “Rule” or “Directive” followed by a dot-decimal sub-identifier; the remaining text is called the headline of the guideline

Amplification

A *typedef* name shall be uniq
the same *typedef* name are c
that *header file* is included in

Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

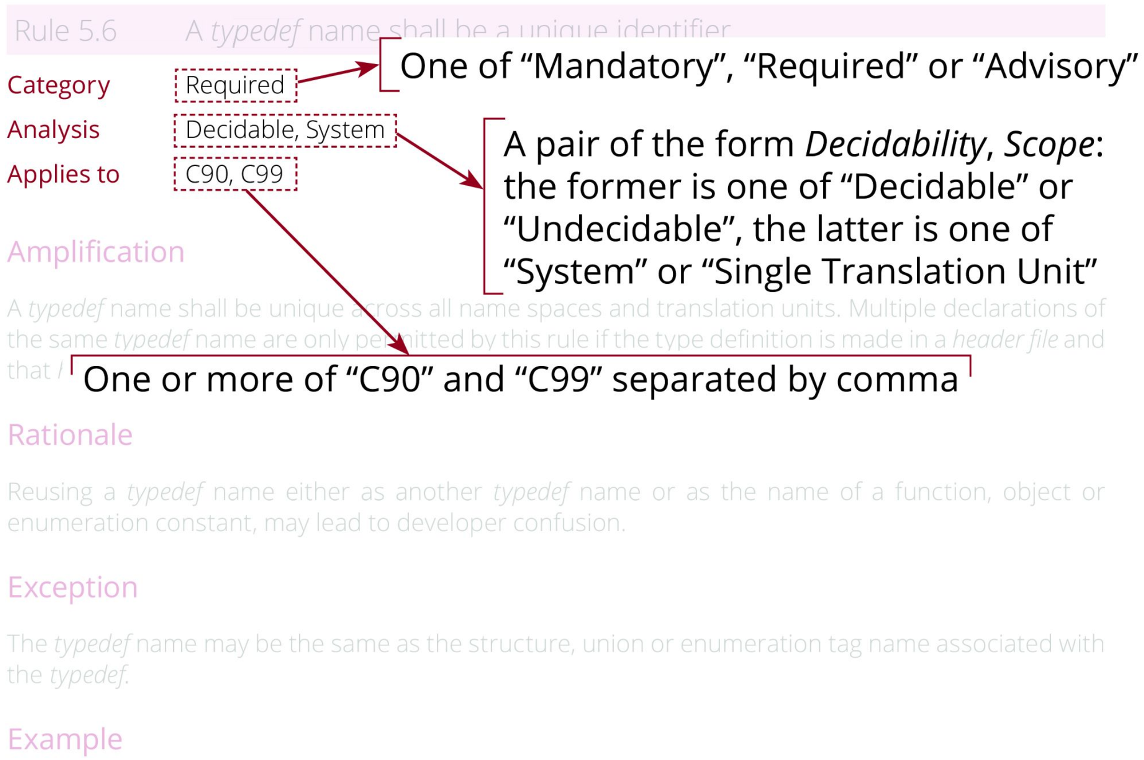
Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

Linaro

Presentation of the MISRA C Guidelines (cont'd)



Rule 5.6 A *typedef* name shall be a unique identifier

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

One of "Mandatory", "Required" or "Advisory"

A pair of the form *Decidability*, *Scope*: the former is one of "Decidable" or "Undecidable", the latter is one of "System" or "Single Translation Unit"

One or more of "C90" and "C99" separated by comma

Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that /

Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

Presentation of the MISRA C Guidelines (cont'd)

Rule 5.6	A *typedef* name shall be a unique identifier
Category	Required
Analysis	De
Applies to	C9

A more precise description of the guideline: this is normative!

Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that *header file* is included in multiple source files.

Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

Presentation of the MISRA C Guidelines (cont'd)

Rule 5.6	A *typedef* name shall be a unique identifier
Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* na nade in a *header file* and that *header file* is inc

The reason why the guideline exists

Rationale

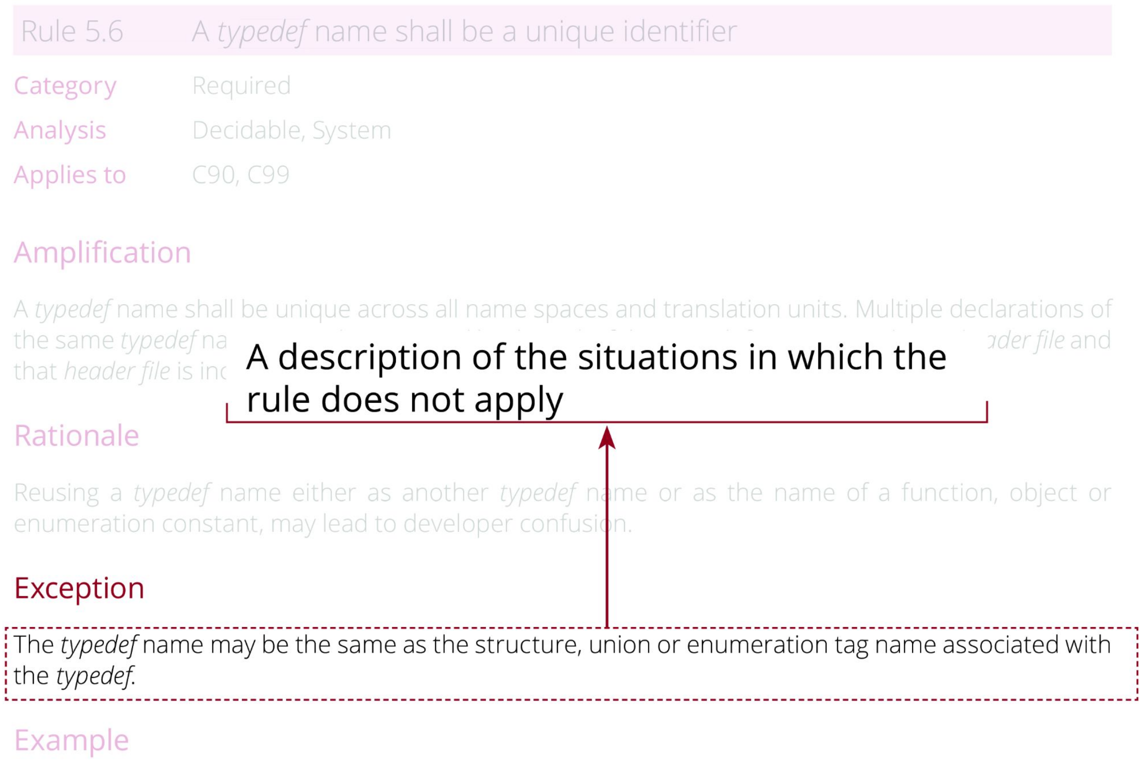
Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

Presentation of the MISRA C Guidelines (cont’d)



Rule 5.6	A *typedef* name shall be a unique identifier
Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* na *ader file* and that *header file* is inc

Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

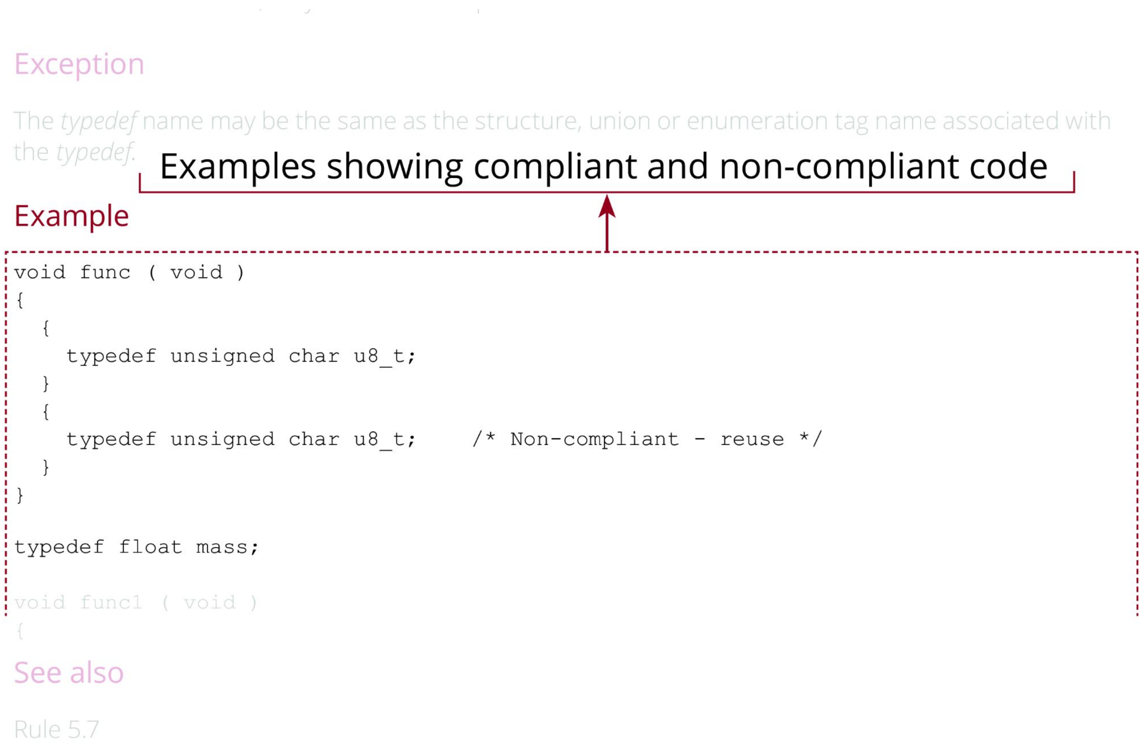
Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

A description of the situations in which the rule does not apply

Presentation of the MISRA C Guidelines (cont’d)



Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Examples showing compliant and non-compliant code

Example

```
void func ( void )
{
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t;    /* Non-compliant - reuse */
  }
}

typedef float mass;

void func1 ( void )
{
```

See also

Rule 5.7

Presentation of the MISRA C Guidelines (cont’d)

Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

```
void func ( void )
{
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t;    /* Non-compliant - reuse */
  }
}

typedef float mass;

void func1 ( void )
{
```

See also

Rule 5.7

Reference to related guidelines

Presentation of the MISRA C Guidelines (cont'd)

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

C90 [Annex G.3], C99 [Annex J.3]

Reference to one or more published sources to be consulted for a fuller understanding of the rationale

Category	Required
Applies to	C90, C99

Amplification

Appendix G of this document lists, for both C90 and C99, those implementation-defined behaviours that:

- Are considered to have the potential to cause unexpected program operation, and
- May be present in a program even if it complies with all the other MISRA C guidelines.

All of these implementation-defined behaviours on which the program output depends must be:

- Documented, and
- Understood by developers.

Note: a conforming implementation is required to document its treatment of all implementation-defined behaviour. The developer of an implementation should be consulted if any documentation is missing.

MISRA C Rule 1.1

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

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99, C11

Linaro

MISRA C Rule 1.1 (cont’d)

Compilers cannot be trusted: they may accept constructs not defined by the language

A conforming compiler does **not need to generate a diagnostic** when a translation limit is exceeded and **an executable may be generated that does not work as expected**

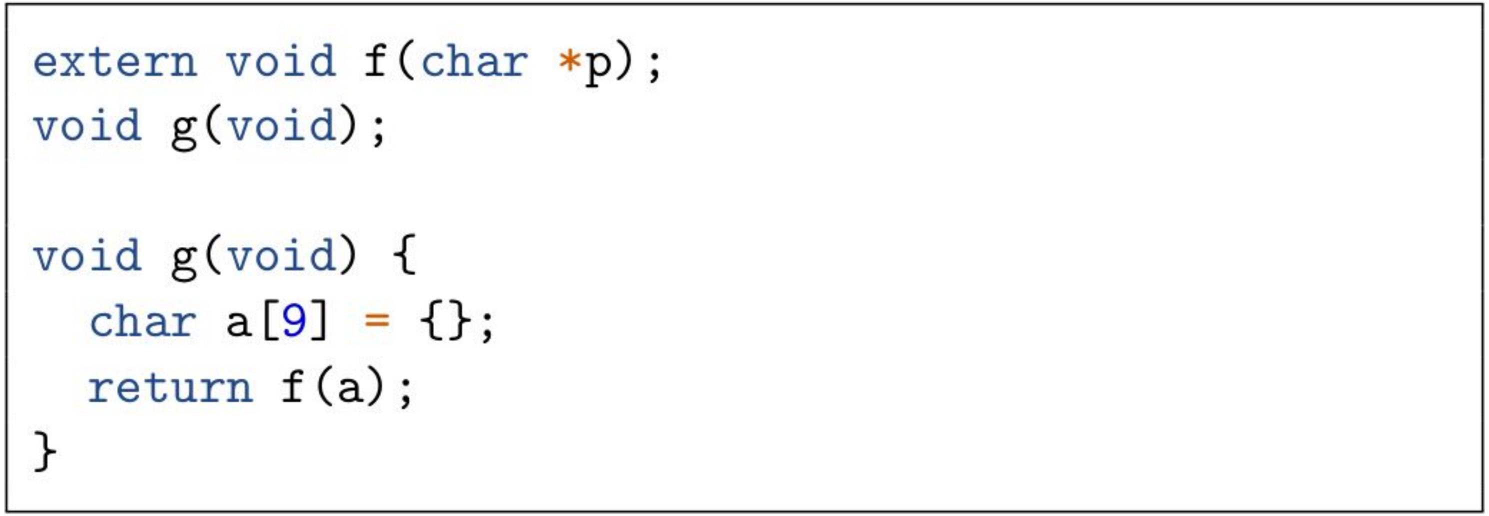
It is possible that some non-conforming compilers **fail to diagnose constraint violations**

Language features that are outside the supported versions of C have **not** been considered when developing the MISRA guidelines

MISRA C Rule 1.1 (cont'd)

Empty initializers and returning of void expressions are **undefined in all versions of the C standard**

GCC accepts them but does **not** document them as extensions



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

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

MISRA C Rule 3.2

Rule 3.2

Line-splicing shall not be used in // comments

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99, C11

MISRA C Rule 3.2 (cont'd)

Rule 3.2

Line-splicing shall not be used in // comments

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();
```

MISRA C Rule 9.1

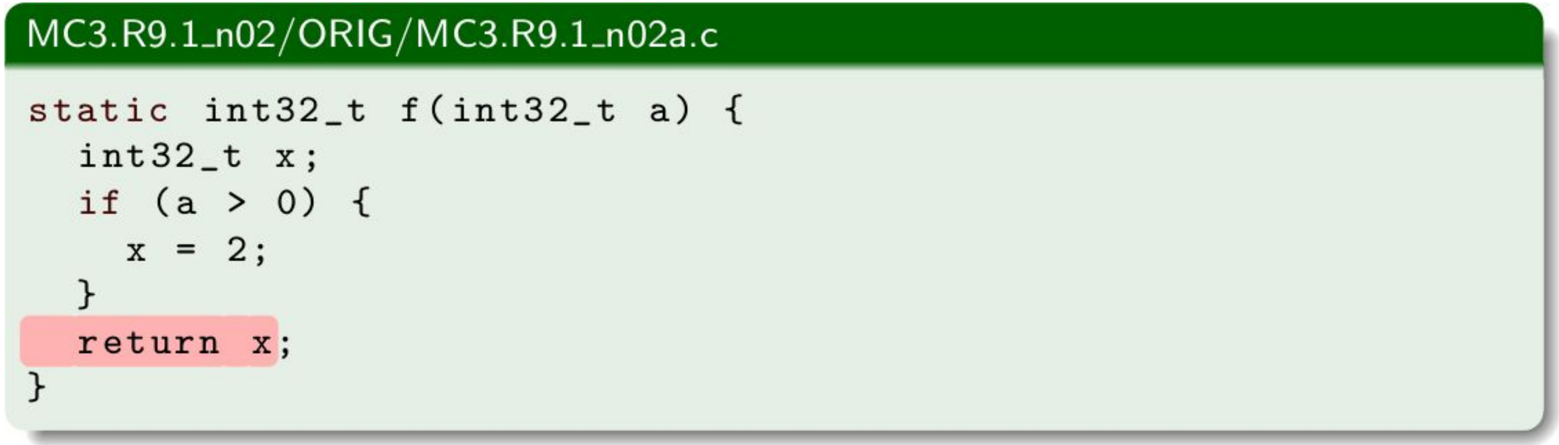
Rule 9.1

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

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99, C11

MISRA C Rule 9.1 (cont’d)

C99 Undefined 10: *the value of an object with automatic storage duration is used while it is indeterminate* (6.2.4, 6.7.8, 6.8)

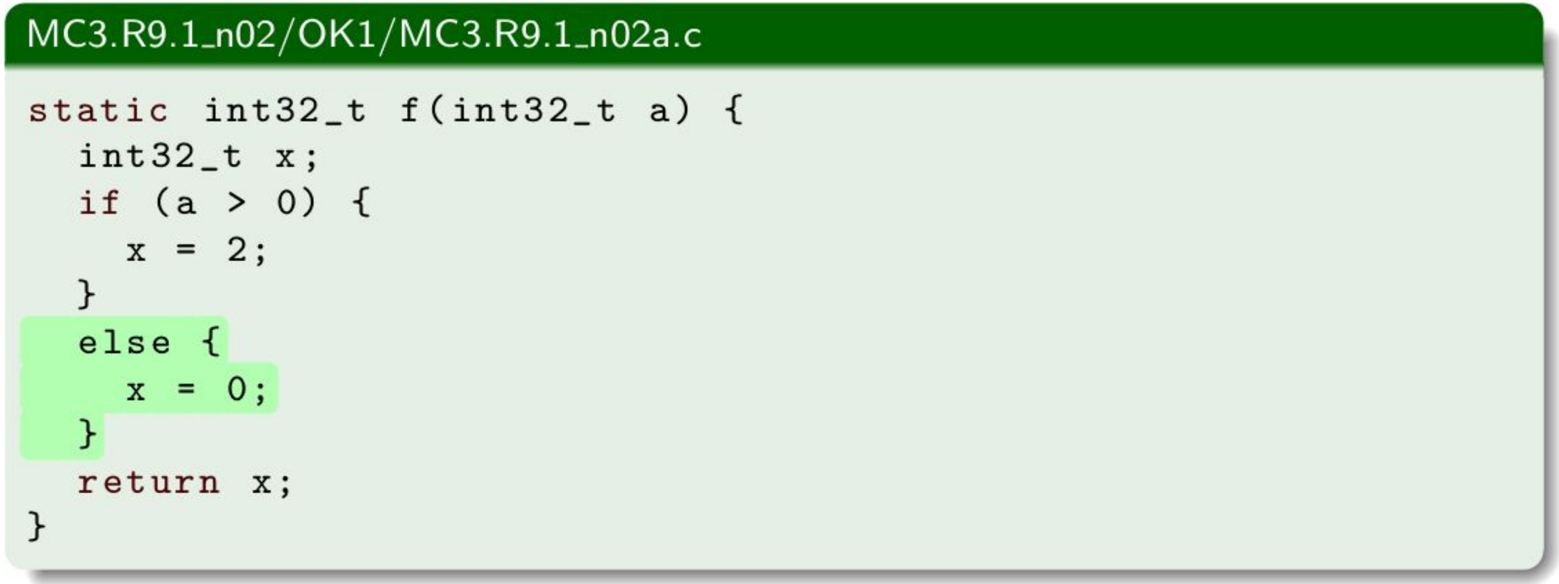


MC3.R9.1_n02/ORIG/MC3.R9.1_n02a.c

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

MISRA C Rule 9.1 (cont'd)

We must ensure that local variables **always** have a value when they are read



MC3.R9.1_n02/OK1/MC3.R9.1_n02a.c

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

MISRA C Rule 13.2

Rule 13.2

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

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99, C11

MISRA C Rule 13.2 (cont'd)

Rule 13.2

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

Between two sequence points the **evaluation order is unspecified**

In addition, the following situations can lead to **undefined behavior**:

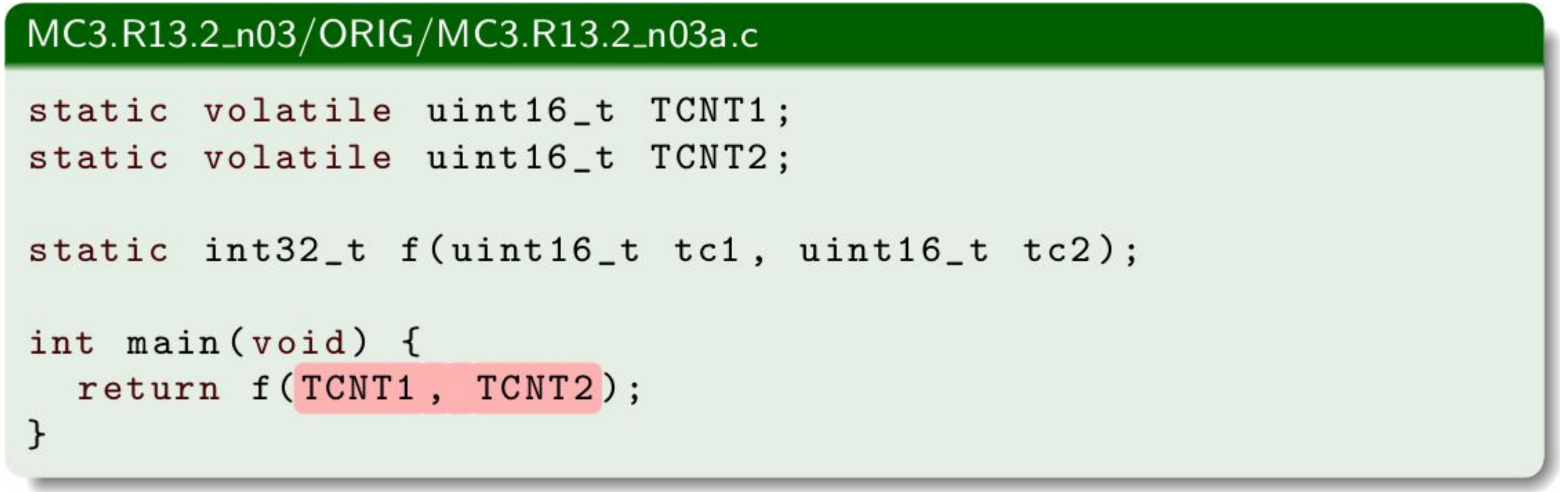
- modifying an object more than once
- modifying and reading an object, unless reading is necessary to store in the object

The logical AND (&&), logical OR (||), conditional (?:) and comma (,) operators have well defined operand evaluation orders

MISRA C Rule 13.2 (cont'd)

TCNT1 and TCNT2 are memory mapped hardware registers

Which side effect is triggered first?


MC3.R13.2_n03/ORIG/MC3.R13.2_n03a.c

```
static volatile uint16_t TCNT1;
static volatile uint16_t TCNT2;

static int32_t f(uint16_t tc1, uint16_t tc2);

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

MISRA C Rule 13.2 (cont'd)

Now the side effect order is definite

MC3.R13.2_n03/OK1/MC3.R13.2_n03a.c

```
static volatile uint16_t TCNT1;
static volatile uint16_t TCNT2;

static int32_t f(uint16_t tc1, uint16_t tc2);

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

ECLAIR Software Verification Platform

General highlights:

- Very **high analysis accuracy**
- Very **high coverage**: 100% of the MISRA C guidelines up to and including Amendment 3
- No configuration required to adapt the analysis to the compilation toolchain and the used compilation options: **automatically detects all the implementation-defined behaviors**, including predefined macros, taking into account all options given to the compiler, assembler, linker, librarian, …
- **Certified 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 Software Verification Platform (cont'd)

More general highlights:

- Many other features besides MISRA checking: metrics, bug finding, stylistic guidelines from BARR-C:2018, integrated requirement management tool
- Automatic verification of **architectural constraints at the software level**, instrumental in providing evidence of **independence/isolation/segregation/freedom from interference**

Highlights **from a CI perspective**:

- All users have access to **fully detailed reports** without installing anything
- All users have access to **private, sophisticated filters** (i.e., locally-stored and independent from one another)
- With the ***ECLAIR Client Kit***, users can **use their favorite IDE** (Eclipse,Visual Studio, Visual Studio Code, NetBeans, CLion, . . .)

Enabling MISRA in OpenCI

Paul Sokolovsky - Linaro Developer Services, TrustedFirmware OpenCI team

Short intro to TrustedFirmware project

https://www.trustedfirmware.org/

TrustedFirmware provides reference implementations of secure software for modern Arm processors, both “A” (application processors) and “M” (microcontrollers).

- Initially, and most notably boot and system services, TrustedFirmware-A and TrustedFirmware-M respectively.
- More projects are added over time, including those of wider interest beyond just Arm community, e.g. mbedTLS.

An OpenSource project, with community consisting largely of Arm chip vendors and system integrators building Arm platforms (e.g. Google).

CI for TrustedFirmware project

Initially, in-house CI at Arm. Was migrated and upgraded to “OpenCI” hosted by Linaro Service group, to improve community access and extend CI coverage and functionality. Largely maintained by Arm and Linaro teams, with growing involvement from wider TrustedFirmware community.

OpenCI consists of:

- Jenkins server to schedule the builds and tests
  - Build agents are Docker and AWS EC2 based
- Linaro LAVA for test execution, both on emulated platforms (FVP, QEMU) and real hardware

TrustedFirmware projects are effectively highly configurable frameworks with a lot of knobs to tweaks - a lot to test. OpenCI runs ~3000 builds daily, and growing. Scalability is one of the biggest tasks.

https://ci.trustedfirmware.org/ | https://tf-ci-users-guide.readthedocs.io/

Static and dynamic analyses

Besides pure builds and tests, OpenCI runs a number of static and dynamic analyses:

- Various style checks as examples of simple static analyses
- Code coverage analysis as an example of dynamic analysis
- Various ad-hoc static analyses for code correctness and avoiding common pitfalls

Another long-standing analysis goal: improve MISRA compliance of the TrustedFirmware projects - the focus of today's presentation.

“Impedance mismatch”

Typically, a goal of MISRA compliance efforts is MISRA certification. The certification applies to a specific product, that is:

- Very specific software project (represented by the exact code tree).
- Very specific hardware platform.
- Very specific configuration.
- Very specific compiler and its options.

All this recorded in a MISRA report, together with “deviations” (exceptions) to MISRA rules.

But that’s not what TrustedFirmware projects are! As was mentioned, they are largely highly configurable (dozens of supported platforms, hundreds of options) frameworks from which specific products can be built.

The aim of MISRA testing for TrustedFirmware

Given the “impedance mismatch” above, the goal of MISRA testing for TrustedFirmware is not achieving certification level itself. But rather:

- Establish and maintain a baseline quality level for projects in regard to the MISRA spec. Roughly speaking, we’d like the codebase to be compliant with all mandatory rules.
- Whenever possible, improve compliance with other MISRA rules (required and advisory) - subject to contribution from the community.
- Provide project members with guidelines and best practices towards achieving MISRA certification, if they choose so.

TrustedFirmware and ECLAIR

ECLAIR is one of the leading MISRA compliance tools on the market. The question is how well it can adapt to “peculiar” TrustedFirmware requirements in that regard.

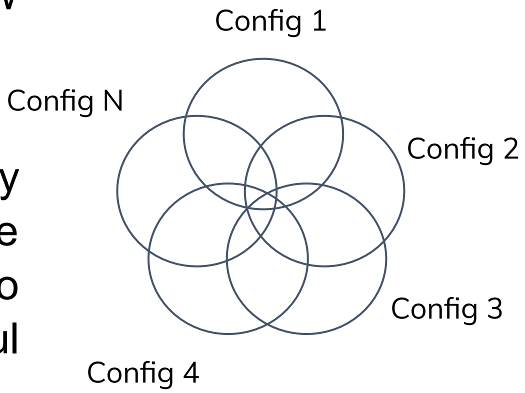
Features which support its usage for this role in TrustedFirmware CI:

- Highly configurable, allows to disable any MISRA rules.
- But the best practice of the tool is not to disable them, but to collect as much information about the codebase as possible. Instead, particular rules can be filtered out at report generation time.
- There can be multiple reports with different filters, e.g. only mandatory rules selected, or also required/advisory.
- Excellent support for running in batch mode, as required for CI.
- Able to produce self-contained browser-based reports.

Major challenge: supporting multiple configs 1/3

As was mentioned previously, TrustedFirmware is a framework with dozens of supported platforms and hundreds of options. For MISRA testing we'd like to get as wide coverage across them as possible, but how to achieve that? The baseline approach is to build each config [among selected for MISRA testing] one by one, and produce a report for each. That's how initial implementation for TF-A was done.

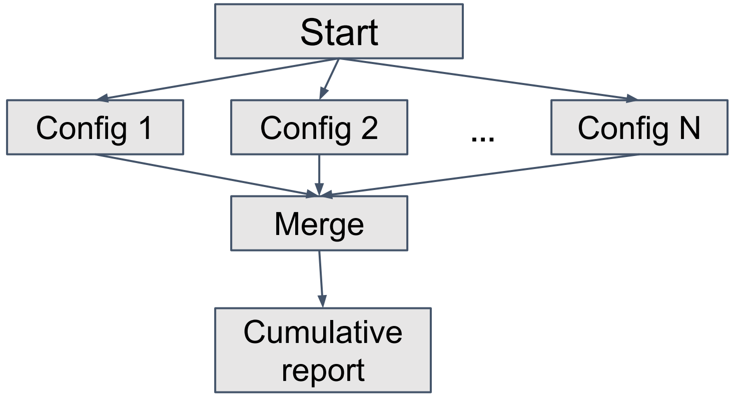
An obvious problem with such an approach is that with already a dozen of configs, it's not very sustainable: the reports are repetitive, with maybe ~90% of content is the same (applying to common code across the configs), so spotting useful differences is almost impossible (or takes high effort).



Major challenge: supporting multiple configs 2/3

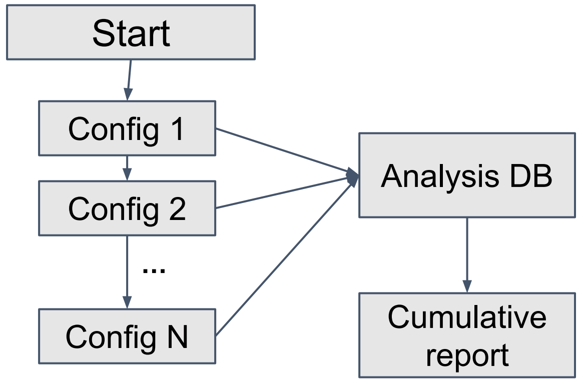
The problem with multiple TF configs is not new to MISRA, it's the same problem as we face with other static/dynamic analyses. In general, there're 2 ways deal with it:

Perform independent analyses on individual configs, and then merge/collate results into a cumulative report.



Example: Code coverage

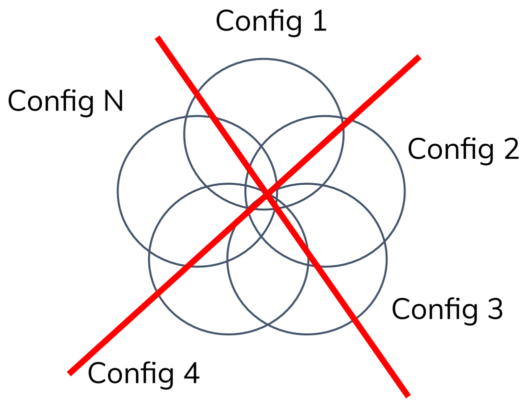
Build multiple configs one by one in the same analysis context, so that results from them would be accumulated "automatically".

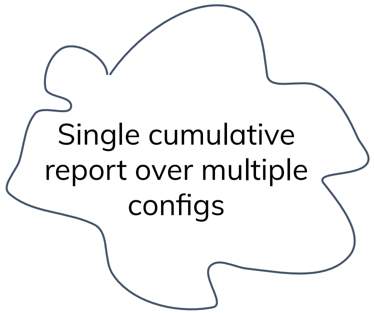


Example: Coverity

Major challenge: supporting multiple configs 3/3

After consulting with BUGSENG, turned out that ECLAIR supports working in 2nd mode, of accumulating successive analyses in a single project database, so any "collation" happens automatically. This approach was tested with TF-M analysis implementation, was found to be successful, and then the plan now is to migrate TF-A analysis to it too.



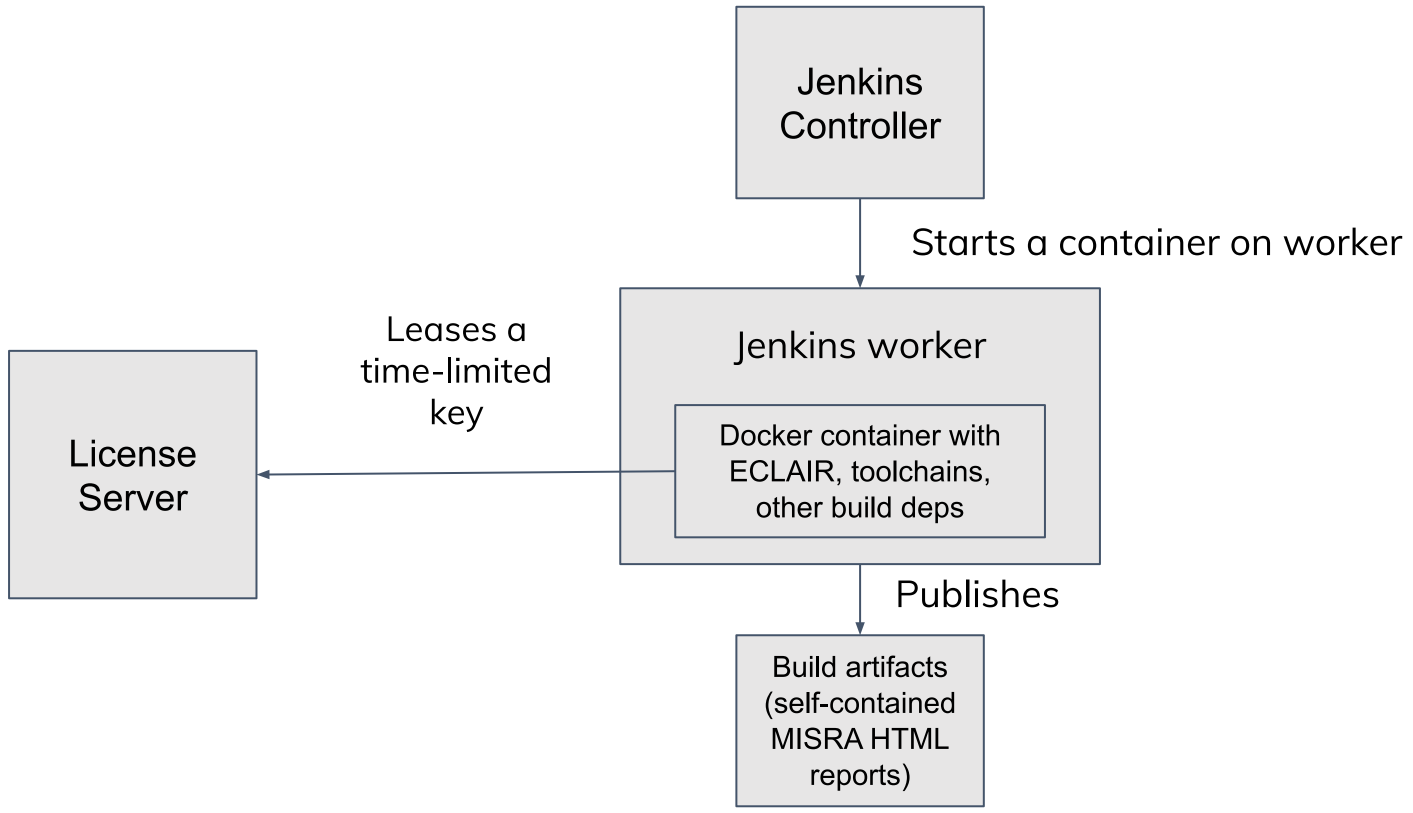


Deploying ECLAIR for OpenCI 1/2

ECLAIR is a proprietary software and requires an active connection to a license server to function. By far, deploying and configuring the license server was the most complex part of the initial setup. And not that it’s really difficult, more just that there are many different deployment options, plus the perceived importance of the license server which is a “gateway” to ECLAIR functionality. That said, after spending on it some time, it works well “in the background”.

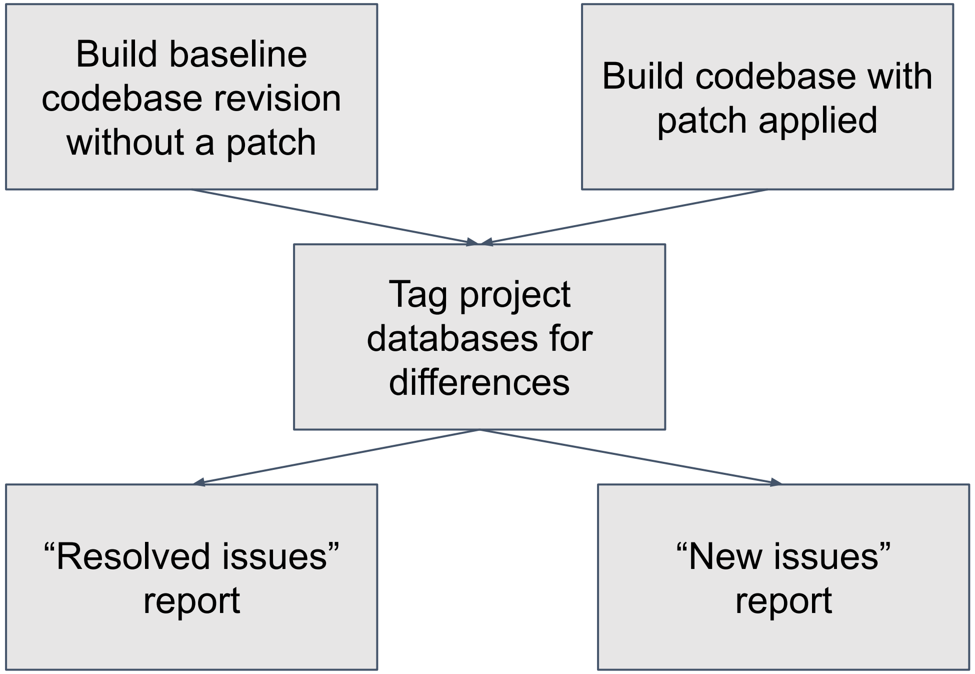
Otherwise, we follow standard build process setup as used in OpenCI: we perform builds in Docker containers, so prepared a Docker image with ECLAIR, toolchains, and other build dependencies preinstalled. On startup, the container requests a time-limited key for the ECLAIR tools, the normal build is performed in the ECLAIR environment, repeated for each requited TF configuration, then individual analysis information is collected in a project database. ECLAIR reporting tool is then run on the project database to produce text and HTML reports, which are then post-processed to make them fully self-contained.

Deploying ECLAIR for OpenCI 2/2

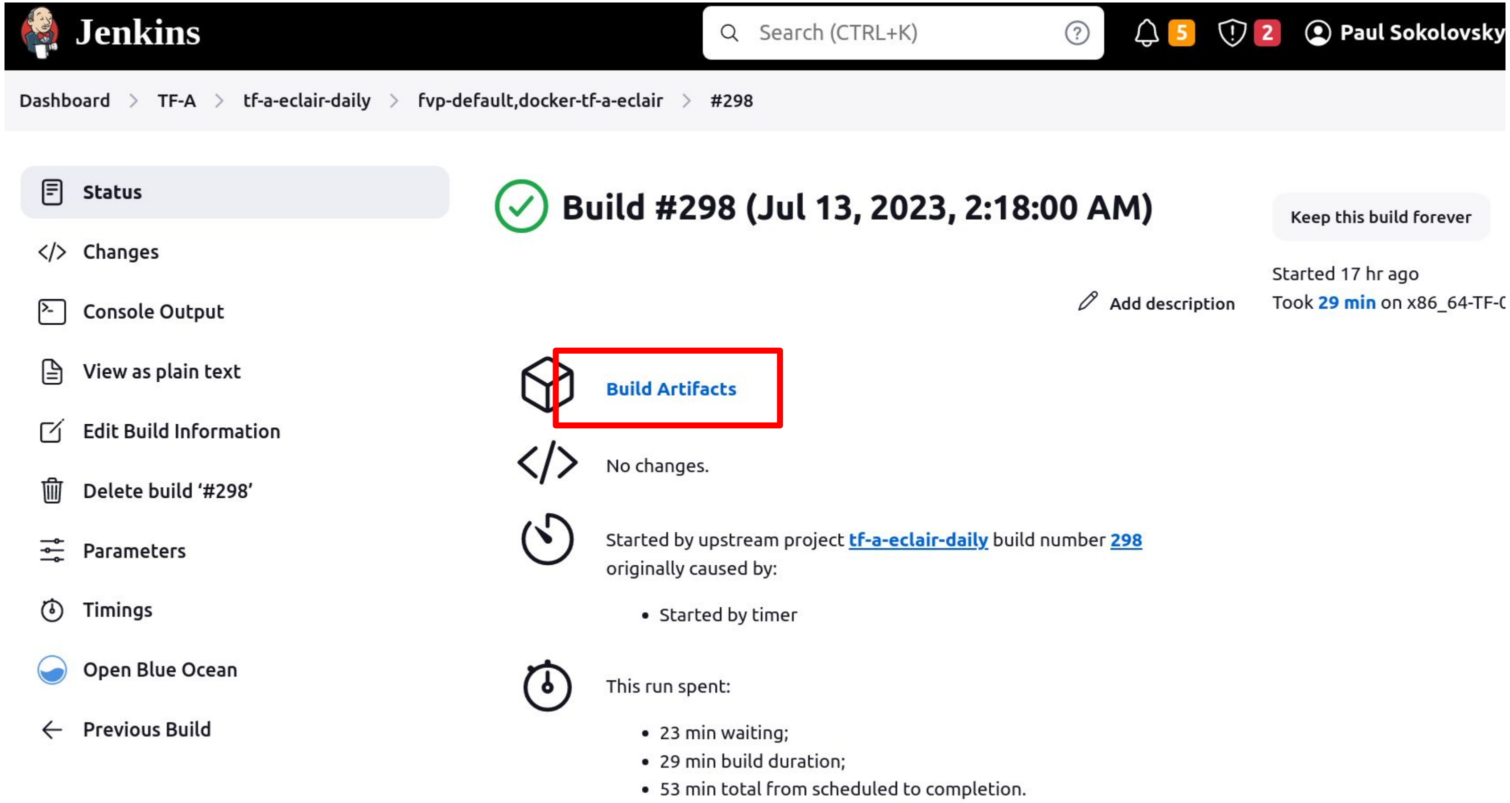

Jenkins Controller
Starts a container on worker
Jenkins worker
Docker container with ECLAIR, toolchains, other build deps
Leases a time-limited key
License Server
Publishes
Build artifacts (self-contained MISRA HTML reports)

Deploying ECLAIR for OpenCI - delta report

Producing delta report for Gerrit patches. All heavy lifting is performed by ECLAIR reporting tools.


Build baseline codebase revision without a patch
Build codebase with patch applied
Tag project databases for differences
“Resolved issues” report
“New issues” report

Example of ECLAIR analysis results



Jenkins

Search (CTRL+K)

5 2 Paul Sokolovsky

Dashboard > TF-A > tf-a-eclair-daily > fvp-default,docker-tf-a-eclair > #298

- Status
- Changes
- Console Output
- View as plain text
- Edit Build Information
- Delete build '#298'
- Parameters
- Timings
- Open Blue Ocean
- Previous Build

Build #298 (Jul 13, 2023, 2:18:00 AM)

Keep this build forever

Started 17 hr ago

Took 29 min on x86_64-TF-(

Add description

Build Artifacts

No changes.

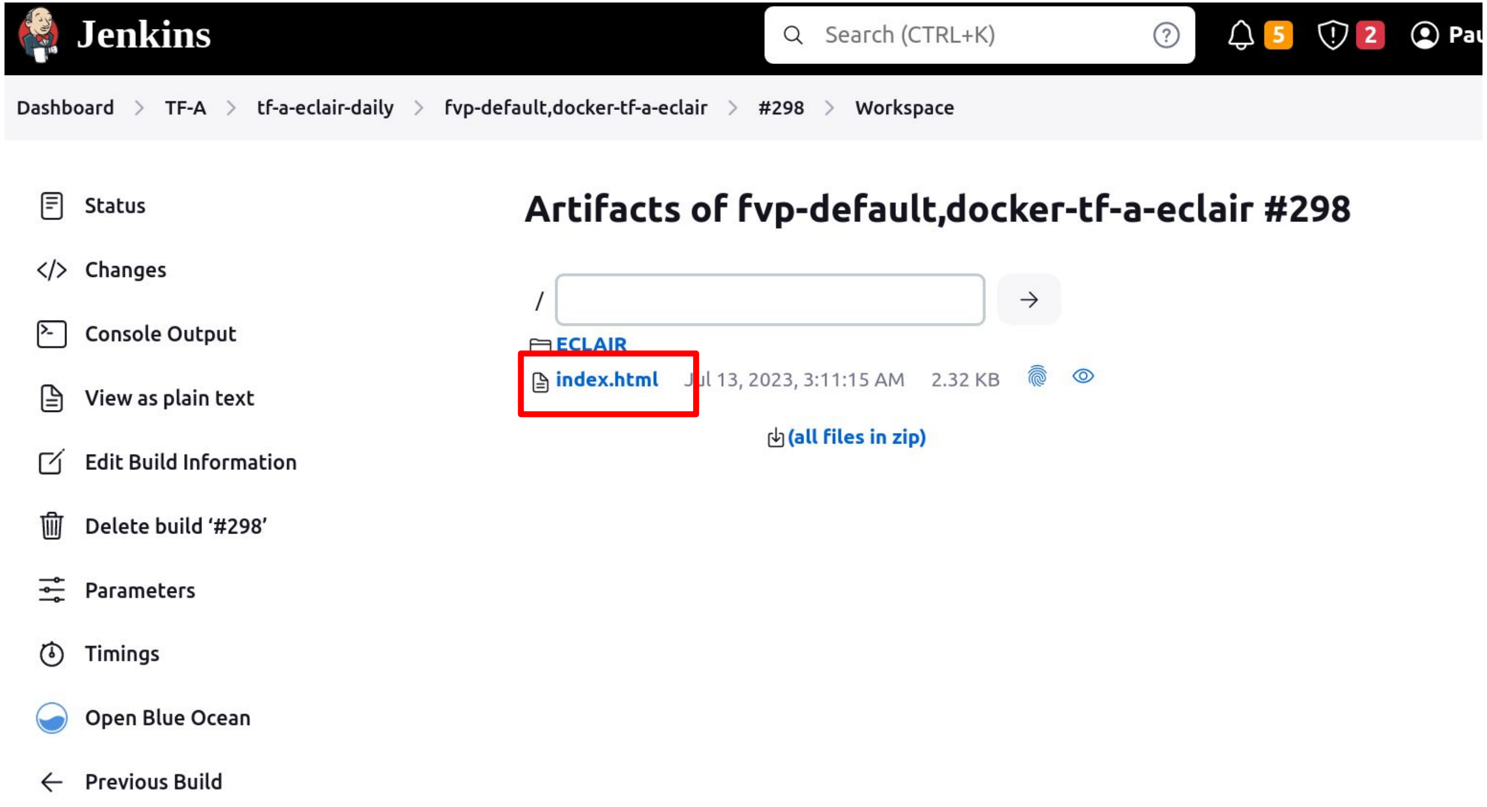
Started by upstream project tf-a-eclair-daily build number 298
originally caused by:

- Started by timer

This run spent:

- 23 min waiting;
- 29 min build duration;
- 53 min total from scheduled to completion.

Example of ECLAIR analysis results


Jenkins
Search (CTRL+K)
5 2 Pau

Dashboard > TF-A > tf-a-eclair-daily > fvp-default,docker-tf-a-eclair > #298 > Workspace

- Status
- Changes
- Console Output
- View as plain text
- Edit Build Information
- Delete build '#298'
- Parameters
- Timings
- Open Blue Ocean
- Previous Build

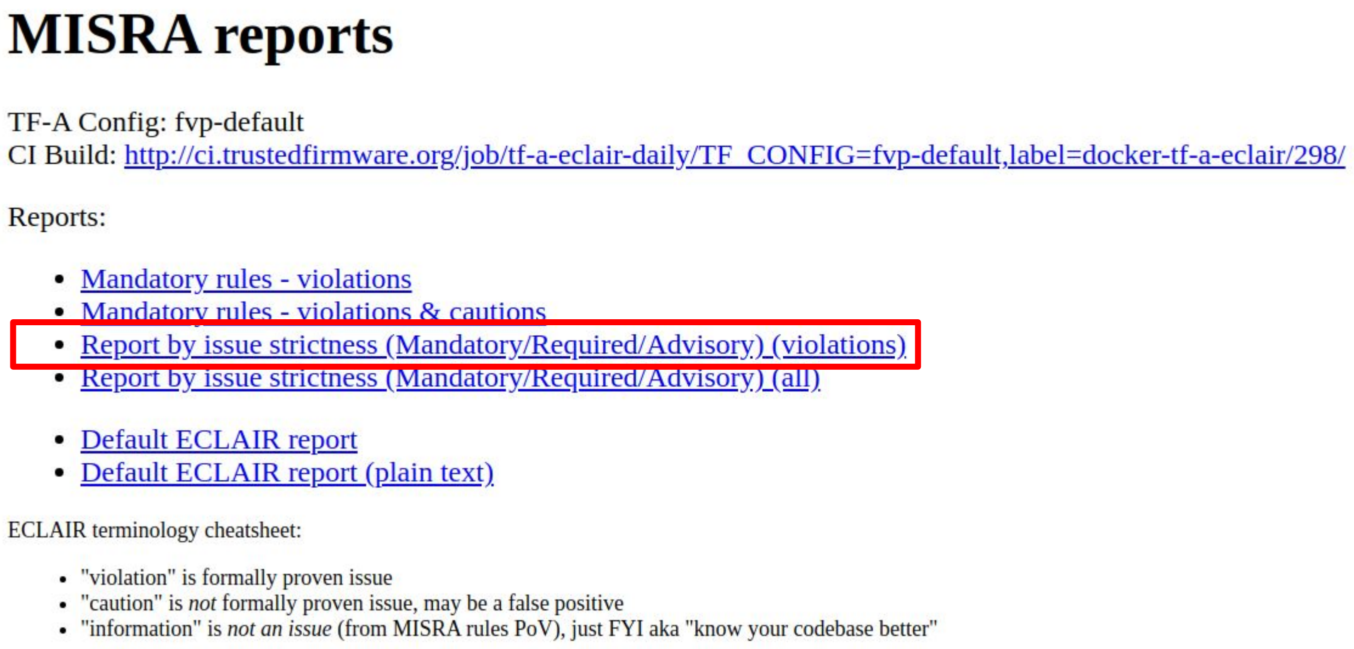
Artifacts of fvp-default,docker-tf-a-eclair #298

/

ECLAIR
index.html Jul 13, 2023, 3:11:15 AM 2.32 KB

(all files in zip)

Example of ECLAIR analysis results



MISRA reports

TF-A Config: fvp-default
CI Build: http://ci.trustedfirmware.org/job/tf-a-eclair-daily/TF_CONFIG=fvp-default,label=docker-tf-a-eclair/298/

Reports:

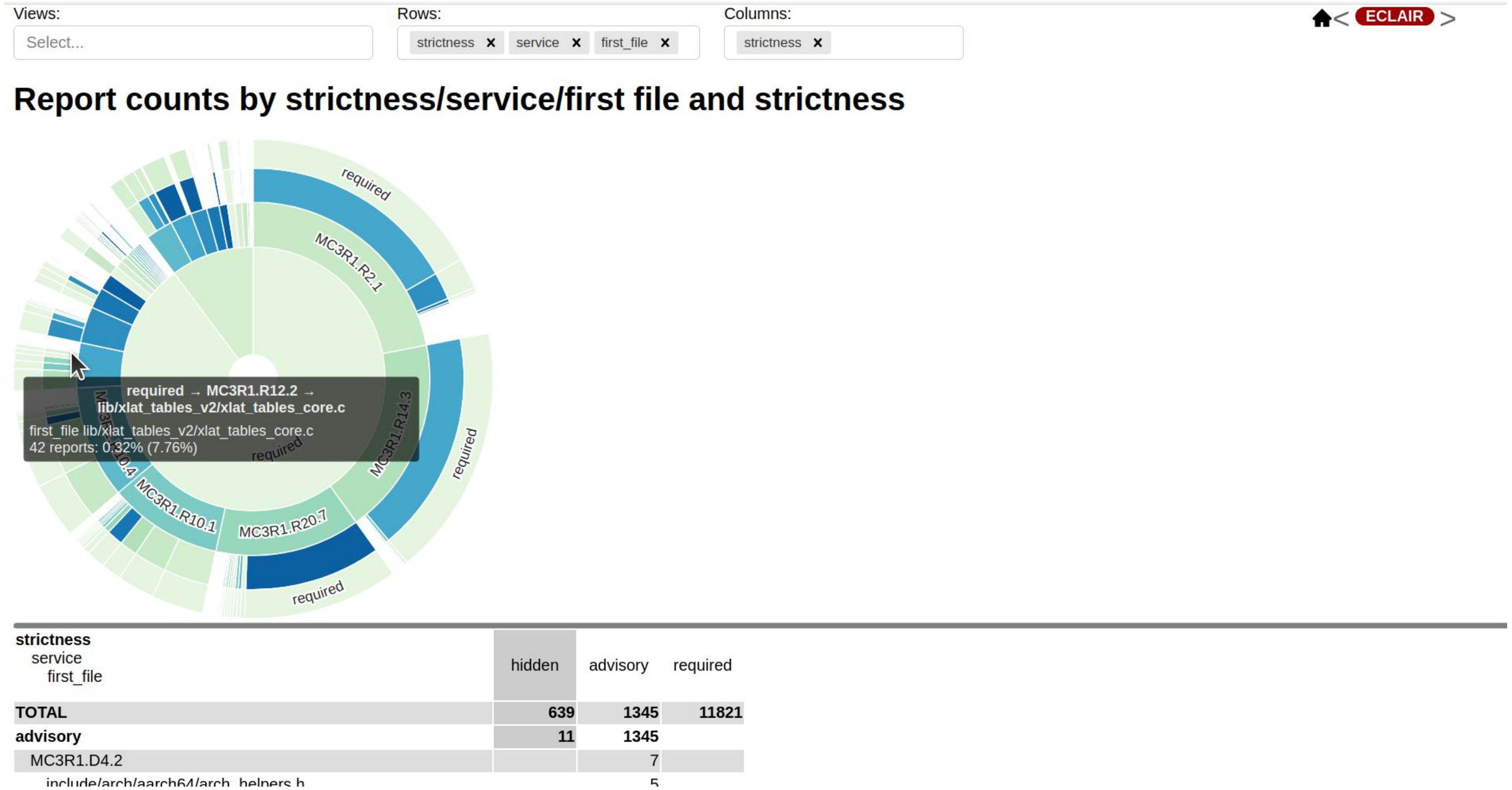
- Mandatory rules - violations
- Mandatory rules - violations & cautions
- Report by issue strictness (Mandatory/Required/Advisory) (violations)
- Report by issue strictness (Mandatory/Required/Advisory) (all)

- Default ECLAIR report
- Default ECLAIR report (plain text)

ECLAIR terminology cheatsheet:

- "violation" is formally proven issue
- "caution" is *not* formally proven issue, may be a false positive
- "information" is *not an issue* (from MISRA rules PoV), just FYI aka "know your codebase better"

Example of ECLAIR analysis results



Views: Select...

Rows: strictness × service × first_file ×

Columns: strictness ×

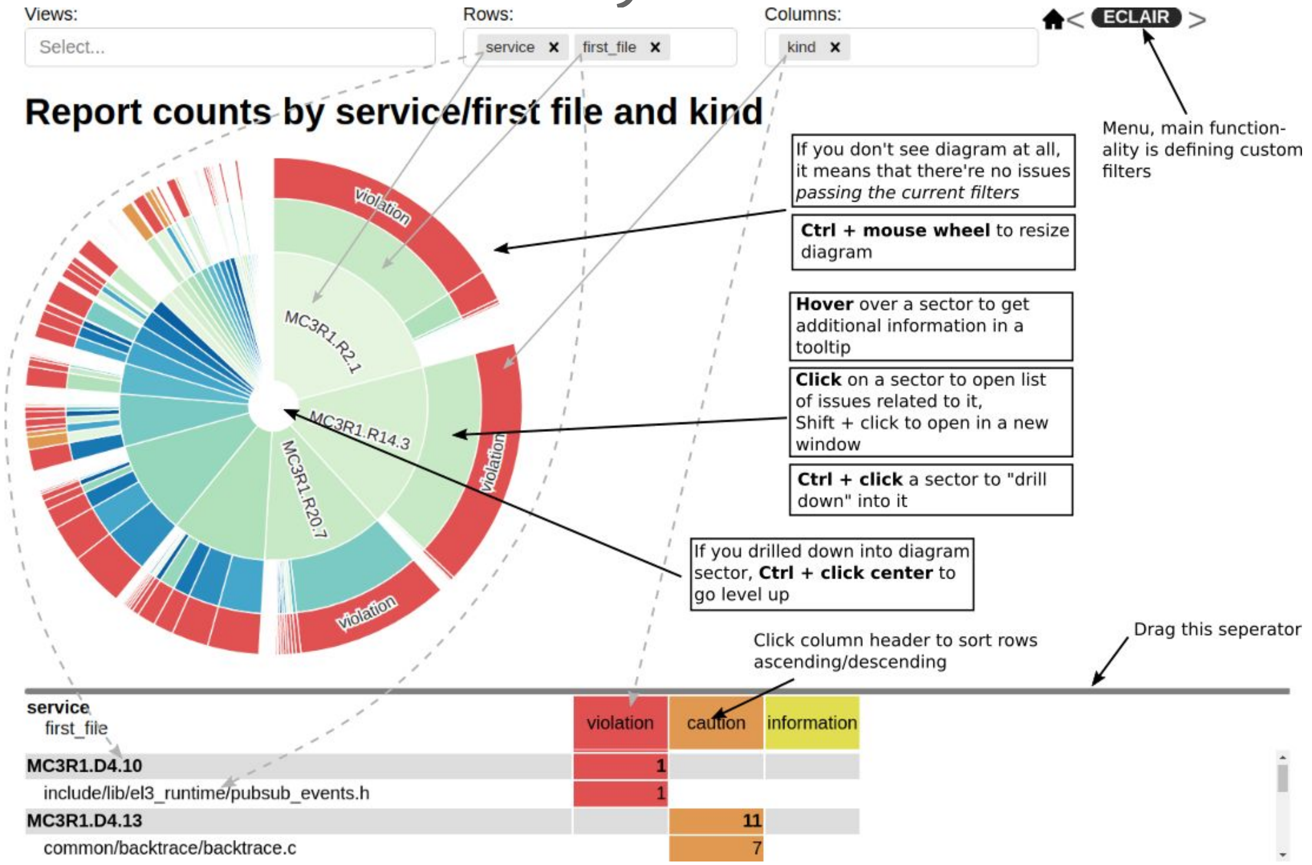
ECLAIR

Report counts by strictness/service/first file and strictness

required → MC3R1.R12.2 → lib/xlat_tables_v2/xlat_tables_core.c
first_file lib/xlat_tables_v2/xlat_tables_core.c
42 reports: 0.32% (7.76%)

strictness service first_file	hidden	advisory	required
TOTAL	**639**	**1345**	**11821**
advisory	**11**	**1345**	
MC3R1.D4.2		7	
include/arch/aarch64/arch_helpers.h		5	

Example of ECLAIR analysis results



Menu, main functionality is defining custom filters

If you don't see diagram at all, it means that there're no issues *passing the current filters*

Ctrl + mouse wheel to resize diagram

Hover over a sector to get additional information in a tooltip

Click on a sector to open list of issues related to it, Shift + click to open in a new window

Ctrl + click a sector to "drill down" into it

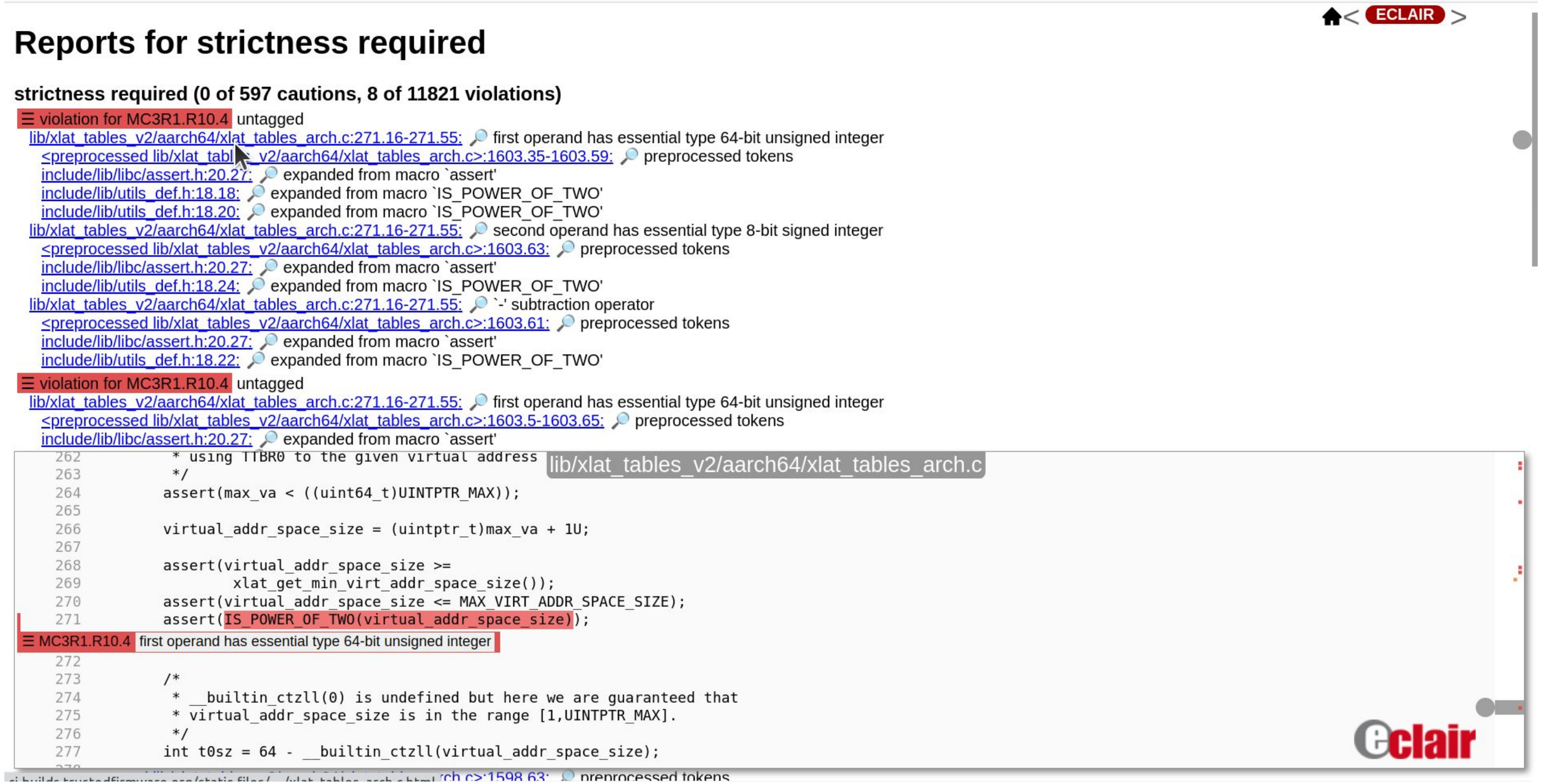
If you drilled down into diagram sector, **Ctrl + click center** to go level up

Click column header to sort rows ascending/descending

Drag this seperator

service first_file	violation	caution	information
MC3R1.D4.10	1		
include/lib/el3_runtime/pubsub_events.h	1		
MC3R1.D4.13		11	
common/backtrace/backtrace.c		7	

Example of ECLAIR analysis results



Reports for strictness required

strictness required (0 of 597 cautions, 8 of 11821 violations)

☰ violation for MC3R1.R10.4 untagged
- lib/xlat_tables_v2/aarch64/xlat_tables_arch.c:271.16-271.55: first operand has essential type 64-bit unsigned integer
  - <preprocessed lib/xlat_tables_v2/aarch64/xlat_tables_arch.c>:1603.35-1603.59: preprocessed tokens
  - include/lib/libc/assert.h:20.27: expanded from macro `assert'
  - include/lib/utils_def.h:18.18: expanded from macro `IS_POWER_OF_TWO'
  - include/lib/utils_def.h:18.20: expanded from macro `IS_POWER_OF_TWO'
- lib/xlat_tables_v2/aarch64/xlat_tables_arch.c:271.16-271.55: second operand has essential type 8-bit signed integer
  - <preprocessed lib/xlat_tables_v2/aarch64/xlat_tables_arch.c>:1603.63: preprocessed tokens
  - include/lib/libc/assert.h:20.27: expanded from macro `assert'
  - include/lib/utils_def.h:18.24: expanded from macro `IS_POWER_OF_TWO'
- lib/xlat_tables_v2/aarch64/xlat_tables_arch.c:271.16-271.55: `-' subtraction operator
  - <preprocessed lib/xlat_tables_v2/aarch64/xlat_tables_arch.c>:1603.61: preprocessed tokens
  - include/lib/libc/assert.h:20.27: expanded from macro `assert'
  - include/lib/utils_def.h:18.22: expanded from macro `IS_POWER_OF_TWO'

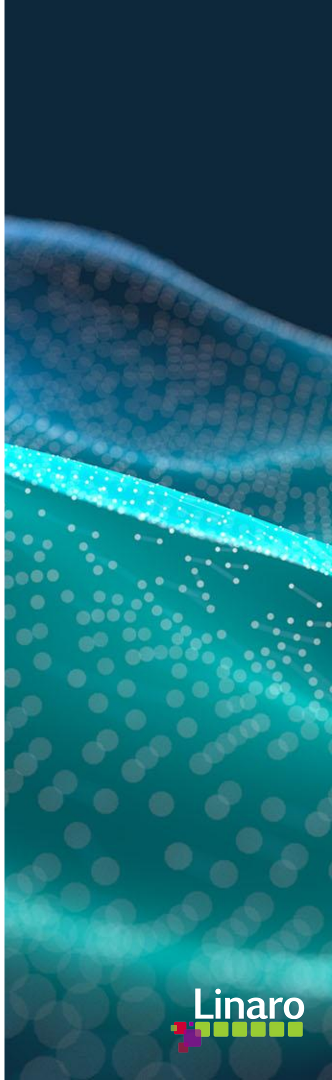
☰ violation for MC3R1.R10.4 untagged
- lib/xlat_tables_v2/aarch64/xlat_tables_arch.c:271.16-271.55: first operand has essential type 64-bit unsigned integer
  - <preprocessed lib/xlat_tables_v2/aarch64/xlat_tables_arch.c>:1603.5-1603.65: preprocessed tokens
  - include/lib/libc/assert.h:20.27: expanded from macro `assert'

lib/xlat_tables_v2/aarch64/xlat_tables_arch.c

```
262          * using TTBR0 to the given virtual address
263          */
264         assert(max_va < ((uint64_t)UINTPTR_MAX));
265
266         virtual_addr_space_size = (uintptr_t)max_va + 1U;
267
268         assert(virtual_addr_space_size >=
269                 xlat_get_min_virt_addr_space_size());
270         assert(virtual_addr_space_size <= MAX_VIRT_ADDR_SPACE_SIZE);
271         assert(IS_POWER_OF_TWO(virtual_addr_space_size));
☰ MC3R1.R10.4 first operand has essential type 64-bit unsigned integer
272
273         /*
274          * __builtin_ctzll(0) is undefined but here we are guaranteed that
275          * virtual_addr_space_size is in the range [1,UINTPTR_MAX].
276          */
277         int t0sz = 64 - __builtin_ctzll(virtual_addr_space_size);
```

rch.c>:1598.63: preprocessed tokens

Q & A



Thank you

About TrustedFirmware.org

TrustedFirmware.org is an open source project implementing foundational software components for creating secure devices. TrustedFirmware 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 code base 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/ for more information.

TrustedFirmware.org is member driven and member funded. To learn more about membership and its benefits, please see the following page or send a request for more information to enquiries@trustedfirmware.org.

About BUGSENG

BUGSENG is a leading provider of solutions and services for static code analysis. Our verification platform has been designed to help engineers develop higher-quality software, effectively, by changing the traditional rules of the game.

To learn more about BUGSENG, please visit our website or email us at info@bugseng.com

Linaro