BeeOS

Static Analysis

Quality and Certification 2017/2018

Author: **Davide Galassi** <davxy@datawok.net>

Quality Supervisor: **Andrea Ceccarelli** <andrea.ceccarelli@unifi.it>

Certification Supervisor: **Lorenzo Falai** <lorenzo.falai@resiltech.com>

Document Version : 1.0.3 (08-04-2018)

Table of Contents

1 Acronyms and Keywords

Keyword	Meaning
POSIX	Portable Operating System Interface
UNIX	A family of multitasking, multiuser OS that derive from AT&T Unix
FOSS	Free and Open Source Software
MISRA	Motor Industry Software Reliability Association
SEI	Software Engineering Institute
CERT	Computer Emergency Response Team

2 Introduction

2.1 Coding Standards

Coding standards are a collection of programming guidelines or requirements that help ensure that the quality of attributes of a project are appropriate to its integrity level.

While standards may be best known for helping programmers meet stringent quality requirements for safety-critical applications, code quality can make or break product success in a wide range of applications. For example, code errors in non-safety-critical automotive systems can result in enormous support costs because of the volumes involved and the overhead for recalls and upgrades, not to mention the product brand devaluation caused by a widely publicized programming defect.

Coding standards protect programmers from errors that can be introduced due to weaknesses in the C or C++ language or from human oversight. Essentially, coding standards provide a foundation for programming best practices.

Compliance with programming standards helps developers:

- Promote portability and avoid unexpected results
- Avoid reliance on compiler or platform-specific constructs
- Identify unreachable or infeasible code, which can indicate a defect that could impact software maintainability
- Prohibit certain language constructs known to be a source of common errors.
- Measurably reduce program complexity
- Improve program testability
- Reduce long-term support costs due to coding errors

While it may be desirable to have strict enforcement of a coding standard, it’s not always practical.

Some products can effectively use a subset to better meet project cost or requirements. For example, the C programming language contains many implementation-defined behaviors that generally should be avoided, such as using the absolute position of bits within a bit-field, but whose use is essential under certain conditions, such as when mapping onto hardware registers.

A coding standard needs to allow some flexibility so that guidelines can be violated in a controlled way. Decisions around flexibility and tradeoffs should be made early enough that deviations can be analyzed and determined and rules established that define appropriate use of that deviation by the project team.

2.2 The Target Project

BeeOS is a FOSS *UNIX-like* operating system focused on simplicity and POSIX compliance.

The overall system is composed by four sub-projects:

- The C standard library (`libc`)
- A utility user-space library (`libu`)
- Some user space applications (`user`)
- The kernel (`kernel`)

This analysis targets the *BeeOS* kernel codebase.

https://github.com/davxy/beeos

The kernel, as most non legacy software, is a “moving target”. Thus, to avoid chasing the codebase with our analysis, we’ve decided to freeze the branch we’re going to analyze at the version 0.1.1 regardless of the evolution of the kernel features during the time taken by the process.

Obviously during the analysis period there can be small changes within the kernel code, but only refactory and fix to get a better compliance with the guidelines.

In the end, any modification that will benefit to the kernel project will be integrated into the upstream kernel codebase (master branch).

2.3 Analysis Scope

The scope of the document is to verify compliance with respect to coding rules from different standards, to analyze differences, and eventually apply corrections on the analyzed code.

In particular, the *BeeOS* sources will be inspected with respect to the following coding rules:

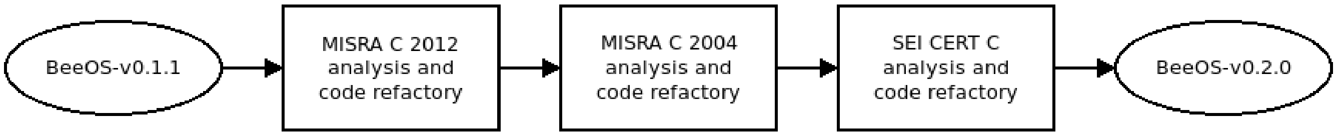
- MISRA C 2012
- MISRA C 2004
- SEI CERT C

Very briefly, the MISRA rules, are a set of coding guidelines that are applied especially in the context of safety critical embedded systems.

The CERT C focus more on prevention of security vulnerabilities like stack-integer and buffer overflows.

The two rules families, when followed together, can provide real rock solid code.

The whole analysis and refactory process is synthesized by the following figure


BeeOS-v0.1.1
MISRA C 2012 analysis and code refactory
MISRA C 2004 analysis and code refactory
SEI CERT C analysis and code refactory
BeeOS-v0.2.0

as evidenced, the process final product will be *BeeOS* v0.2.0.

For every coding standard we will provide a compliance matrix, that is a table where for each rule the violations are counted to give a quantitative result.

For each coding standard matrix the *pre-intervention* column refers to the violations found with the previous phase output; the *post-intervetion* column is produced using the output of the **whole** process (kernel v.0.2.0).

The choice to don’t take as the post-intervention values the output of the specific phase is due to the fact that some violations that are still present in a phase output may be resolved (or introduced) as a side effect of some refactory required by a coding standard that follows in the chain. Thus, to give a better idea of the overall work result, we’ve decided to provide a matrix containing the output of the process viewed as a single functional block.

2.4 Codebase Metrics

	Pre-Intervention v0.1.1	**Post-Intervention v0.2.0**
Files	110	112
Functions	249	255
Blank Lines	1,423	1,576
Preprocessor Lines	760	791
Code Lines	4,841	4998
Comment Lines	3,811	3,894
Comment to Code Ratio	0.79	0.78
Declarative Statements	1,299	1,297
Executable Statements	2,213	2,370
Inactive Lines	51	55
Lines	10,302	10,735

3 MISRA C

3.1 Introduction

MISRA C is a set of software development guidelines for the C programming language developed by the Motor Industry Software Reliability Association. Its aims are to facilitate code **safety**, **security**, **portability** and **reliability** mainly in the context of embedded systems, specifically those systems programmed in C89 and C99.

The guidelines define a **subset** of the C language in which the opportunity to make mistakes is either removed or reduced.

The standard is divided in two macro areas:

- **Directives:** guidelines for which it is not possible to provide the full description necessary to perform a check for compliance. Static analysis tools may be able to assist in checking compliance with directives but different tools may place widely different interpretations on what constitutes a non-compliance.
- **Rules:** guidelines for which a complete description of the requirement has been provided. It should be possible to check that source code complies with a rule without needing any other information.

Furthermore to every MISRA C rule and directive is given a single category of:

- **Mandatory:** violations to a mandatory guideline is not permitted.
- **Required:** formal deviation documentation is necessary for required guidelines violations.
- **Advisory:** these are recommendations, formal deviations documentation are not necessary.

Note that the status of “advisory” does not mean that these items can be ignored, but rather that they should be followed as far as is reasonably practical.

In some instances it may be necessary to **deviate** from the guidelines. Deviations are divided into two categories:

- **Project deviation:** used when a guideline is deviated in a particular class of circumstances.
- **Specific deviation:** used for deviations that interest a single instance in a single file.

<u>In this assessment we’re not aiming for strict compliance.</u>

Follows that we’ll encounter several deviations to required guidelines where the project requires it both for design or practical reasons.

3.2 Static Analysis

It is possible to check that C source code complies with MISRA C by means of inspection alone. However, this is likely to be extremely time-consuming and error prone.

Any realistic process for checking code against MISRA C will therefore involve the use of at least one **static analysis tool**.

Unfortunately, maybe because this type of analysis is more frequently performed in a professional industrial engineering context, there is a lack of free tools in this area. Anyway we were able to successfully conduct our analysis by combining the evaluation versions of two non-free tools and by cheating a bit.

The used tools are:

- ***Understand*** from *Scientific Toolworks*.
- ***PC-Lint*** from *Gimpel Software*

Scientific Toolworks gives the opportunity to download an evaluation version of the complete *Understand* software. The tool looks very solid but the evaluation is only able to analyze more or less the 70% of the complete MISRA rules set.

PC-Lint offers an on-line evaluation page capable to analyze every other rule

```
http://www.gimpel-online.com/OnlineTesting.html
```

Its usage is a bit less practical since they offer only a web form where the code to be analyzed should be pasted and then sent to the backing engine for analysis.

The code that we paste in the form should be "*self-contained*", meaning that it cannot contain external include dependencies (as our code obviously has). To solve the issue is sufficient to *preprocess* the source file to expand, in a single translation unit, every dependency. For reference we give the used command:

```
$ gcc -P -E -CC -Ilibc/include -Ikernel/src -Ikernel/include kernel/src/mysource.c

-P  : inhibit the generation of line markers in the preprocessor output
-E  : preprocess only
-CC : do not dicard comments, including during macro expansion
-I  : include path
```

We've exploited such a page to analyze, file after file, our entire kernel codebase (obviously using an *ad-hoc* script that does all the work in the background for us).

The *PC-Lint* hack filled the gap and we were able to analyze the kernel code against the full MISRA C rule-set.

As a C toolchain we've mainly use the free **GNU gcc** and **binutils.**

4 MISRA C: 2012

4.1 Brief

In 2013, MISRA C 2012 was announced. MISRA C 2012 extends its predecessor with support to the C99 version of the C language (while maintaining guidelines for C89) and including a number of improvements that can reduce the cost and complexity of compliance, whilst aiding consistent, safe use of C in critical systems.

MISRA C 2012 contains **16 directives** and **143 rules**.

4.2 Directives

4.2.1 - Implementation

Directive 1.1 (Required) - *Any implementation-defined behavior on which the output of the program depends shall be documented and understood.*

To increase portability across compilers and architectures, constructs that are defined as “implementation defined” or “unspecified” by the ANSI C standard are avoided.

4.2.2 - Compilation and build

Directive 2.1 (Required) - *All source files shall compile without any compilation errors.*

Enforced by the “`-Wall`” and “`-Werror`” compiler flags.

4.2.3 - Requirements traceability

Directive 3.1 (Required) – *All code shall be traceable to documented requirements.*

There is only core one requirement: compliance with the POSIX standard.

4.2.4 - Code design

Directive 4.1 (Required) – *Run-time failures shall be minimized.*

Every function return value is checked. In the eventuality of a runtime error the work already done by the failing function is subject to a rollback to resume the previous state.

Project deviation

For efficiency, sanity checks are done only for input values coming from user-space. Inter-kernel function arguments are not subject to sanity check when argument validity is a function precondition constraint.

Directive 4.2 (Advisory) – *All usage of assembly language should be documented.*

Low level assembly language is used internally and in very specialized contexts, for example to implement the entry gate of interrupt handling routines or kernel low level startup code. Documentation is such very self contained cases is not useful.

Documentation has been inserted for assembly functions that may be interesting for driver developers (e.g. x86 port i/o or enable/disable interrupts).

Directive 4.3 (Required) – *Assembly language shall be encapsulated and isolated.*

Violations were present. Inline assembly code was present directly within the functions body were necessary (e.g. task context switch or page table swap).

Assembly code has been isolated and properly encapsulated within C macros or stand-alone assembly source files.

Directive 4.4 (Advisory) – *Sections of code should not be "commented out"*

Unused code is reduced to the minimum. In the eventuality that some code must be excluded from compilation the "`#if 0 #endif`" trick is used. This is compliant with the MISRA requirements.

Directive 4.5 (Advisory) – *Identifiers in the same name space with overlapping visibility should be typographically unanbiguous*

Depending upon the font used to display the character set, it is possible for certain glyphs to appear the same. This has been generally avoided in the project.

Specific deviation

The tool warn us that two members of the same structure, `c_iflag` and `c_lflag`, are typographically similar and thus can be source of future errors.

Unfortunately there is nothing to do here, the two fields are members a POSIX defined structure, `termios`. Compliance with POSIX is paramount.

Directive 4.6 (Advisory) – *`typedefs` that indicate size and signedness should be used in place of the basic numerical types*

This directive is strictly followed just where there is a valid reason to do so. In particular, within the lowest level code and data structures where the size and the elements alignment is fundamental for the system working.

For example, the interrupt descriptor table entries:

```
struct idt_entry {
    uint16_t offset_lo; /* Lower 16 bits of the addr to jump to */
    uint16_t selector;  /* Kernel segment selector. */
    uint8_t  zero;      /* This must always be zero. */
    uint8_t  flags;     /* More flags. See documentation. */
    uint16_t offset_hi; /* Upper 16 bits of the addr to jump to. */
};
```

Project deviation

If there isn't an integral type width requirement a deviation is introduced.

Directive 4.7 (Required) – *If a function returns error information, then that error information shall be tested*

Respected by every caller.

An ignored bad return value, in the kernel context, can jeopardize the whole system stability and, at best, trigger a *kernel-panic* and a machine reset.

Before the intervention, a safe deviation causing 47 violations was present. The `kprintf` function, i.e. the kernel equivalent of `printf`, return value (i.e. the number of characters effectively printed) was always ignored. The function prototype has been modified to return `void`.

Directive 4.8 (Advisory) – *If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.*

Project deviation

Opaque pointers are not used by the project. Structures that are used across translation units are inserted within a header file to allow stack or global allocation.

Types that are used only within a translation unit are generally defined within the translation unit itself, thus hidden to the external world.

Directive 4.9 (Advisory) – *A function should be used in preference to a function-like macro where they are interchangeable.*

Effective inlining of `inline` functions are left to the compiler. Thus, is not guaranteed that `inline` functions will, in the end, not result in a stand-alone function call.

Project deviation

For maximum efficiency, small code snips are defined as function-like macros.

Directive 4.10 (Required) – *Precautions shall be taken in order to prevent the contents of a header file being included more than once.*

Every header file starts with the well known preprocessor guards to explicitly avoid double inclusion.

```
#ifndef HEADER_NAME_
#define HEADER_NAME_
  ...
#endif /* HEADER_NAME_ */
```

Directive 4.11 (Required) – *The validity of values passed to library functions shall be checked*

Validity of parameter arguments are always checked by the caller. Internal functions assume that the input arguments constraints are respected.

Directive 4.12 (Required) – *Dynamic memory allocation shall not be used*

If a decision is made to use dynamic memory, care shall be taken to ensure that the software behaves in a predictable manner.

Project deviation

The tool indicates zero violations, but this is a false negative. The kernel allocates dynamic objects (files, inodes, dentries, etc.) via its own memory allocator.

As a general purpose kernel, a priori knowledge of the required objects number is not possible. Pre-allocation of a fixed number of objects in global memory is possible but not a desirable feature in a system like this.

Directive 4.13 (Advisory) – *Functions which are designed to provide operations on a resource should be called in an appropriate sequence*

Every function that acquire resources is paired with the dual one to release them. They are called in the appropriate sequence (e.g. `iget/iput, kmalloc/kfree,` etc.).

If a complex structure has nested resources to be allocated they are released in the opposite order followed by the initialization.

In case of errors in the middle of a function that allocates several resources a proper rollback sequence is guaranteed.

4.3 Rules

4.3.1 - Standard C environment

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

Generally, standard C89 syntax has been followed, with no constraints violations.

Rule 1.2 - (Advisory) *Language extensions should not be used.*

No compiler defined language extensions were used.

Project deviation

The `asm` keyword, used to insert inline assembly code, is not in the ISO C standard (n1570 draft of C2011), but is mentioned as a common extension in annex J (common extensions).

Rule 1.3 - (Required) *There shall be no occurrence of undefined or critical unspecified behavior.*

Expressions marked as "undefined" or "unspecified" behavior by the standard were avoided.

4.3.2 - Unused code

Rule 2.1 (Required) - *A project shall not contain unreachable code.*

Coverage testing has not been performed. But we are confident that the project doesn't contains unreachable code.

The tool signals zero violations, but it is not very reliable with this type of check.

Rule 2.2 (Required) - *There shall be no dead code.*

The project doesn't contains unreachable code.

False positives

The tool identified 38 violations, all of which are code statements called from assembly language code.

Rule 2.3 (Advisory) - *A project should not contain unused type declarations.*

Specific deviation

The tool reports the `elf_section_header` structure, defined within the `elf.h` header file, as unused. Indeed that structure has been defined for future usage (i.e. may be required for a stack trace on kernel panic).

Rule 2.4 (Advisory) - *A project should not contain unused tag declarations.*

The same violation of Rule 2.3 has been reported.

Rule 2.5 (Advisory) - *A project should not contain unused macro declarations.*

False positives

Macros used within assembly language code are not detected by the tool.

Project deviation

Some macro, especially the ones defining some device register bit values or flags, are defined for completeness or future usage.

Rule 2.6 (Advisory) - *A function should not contain unused label declarations.*

Compliant.

Rule 2.7 (Advisory) - *There should be no unused parameters in functions.*

Project deviation

Some POSIX defined functions prototypes requires some parameter list that, at this development stage, are not used by the kernel implementation and are silently ignored.

For example, within the `mount` syscall implementation

```
int sys_mount(const char *source, const char *target, const char *fstype,
              unsigned long flags, const void *data);
```

only the `source`, `target` and `fstype` parameters are used.

4.3.3 - Comments

Rule 3.1 (Required) - *The character sequences `/*` and `//` shall not be used within a comment.*

The tool indicates 114 violations and all of them should be ignored.

The issue is caused by URLs embedded within the comments. The most common (114 violations and 112 source files) is caused by the license header that is present in every source file.

```
/*
 *    .... <http://www.gnu/licenses/>
 */
```

Rule 3.2 (Required) - Line-splicing shall not be used in // comments.

C99 `//` comments are avoided in favor of the C89 `/* */` comment style.

4.3.4 - Character sets and lexical conventions

Rule 4.1 (Required) - *Octal and hexadecimal escape sequences shall be terminated.*

Compliant.

Rule 4.2 (Advisory) - *Trigraphs should not be used.*

Compliant.

4.3.5 - Identifiers

Rule 5.1 (Required) - *External identifiers shall be distinct*

The C89 is too restrictive, requiring that the first 6 characters of the external identifiers to be distinct. C99 increase the number to 31.

We are compliant to the C99 rule version.

Rule 5.2 (Required) - *Identifiers declared in the same scope and name space shall be distinct.*

Compliant to both the C89 (31 chars) and the C99 (63 chars) versions.

Rule 5.3 (Required) - *An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.*

Some violations were found by the tool and were fixed for compliance.

In particular variables hiding a global scope structure name

```
struct bufctl {  … };

void func(struct bufctl *bufctl) { … }
```

Rule 5.4 (Required) - *Macro identifiers shall be distinct.*

Compliant.

Rule 5.5 (Required) - *Identifiers shall be distinct from macro names.*

There were two violations because of this code snip

```
#ifndef __ASSEMBLER__
 static inline void *phys_to_virt(void *addr)
 {
     return (char *)addr + KVBASE;
 }
#else
 #define phys_to_virt(addr) ((addr) + KVBASE)
#endif
```

and, similarly, the `virt_to_phys` identifier.

The macro version was used only in one place thus has been removed and substituted with a direct calculation.

Rule 5.6 (Required) - *A* `typedef` *name shall be a unique identifier.*

Compliant.

Rule 5.7 (Required) - *A tag name shall be a unique identifier.*

There were four violations all caused by structures forward declarations.

Even though a forward declaration should be an accepted construct, we’ve taken a corrective action to attain compliance.

Typical forward declarations usage:

```
struct inode;

struct inode_ops {
    ...
    int (*inode_read)(struct inode *inod, char *buf, size_t n);
    ...
};

struct inode {
    ...
    struct inode_ops *ops;
};
```

Rule 5.8 (Required) - *Identifiers that define objects or functions with external linkage shall be unique.*

There were two violations. Both caused by a global variable, with external linkage, with the same name of a structure field. One case:

```
struct ramdisk {
   void  *addr;
   size_t size;
};

struct ramdisk ramdisk;
```

Replaced by:

```
struct {
    void  *addr;
    size_t size;
} ramdisk;
```

Rule 5.9 (Advisory) - *Identifiers that define objects or functions with internal linkage should be unique.*

There were two violations. The structures names were the same as the structures instances (similar to violation of Rule 5.8, but this time the instances has internal linkage).

Easily resolved by renaming the structures.

4.3.6 - Types

Rule 6.1 (Required) - Bit-fields shall only be declared with an appropriate type.

Bit-fields are not used within the project. Preference was given to the more traditional macro flags and masks.

Rule 6.2 (Required) - *Single-bit named bit fields shall not be of a signed type.*

Refer to MISRA C 2012 Rule 6.1.

4.3.7 - Literals and constants

Rule 7.1 (Required) - *Octal constants shall not be used.*

PC-Lint reports 49 violations but for the *Understand* tool the code is compliant.

The non congruence is probably due to the fact that *Understand* doesn't expand macros that are not part of the project or it doesn't consider a macro value an *"octal constant"*.

Project deviation

These violations are just ignored because:

- The macro values are not part of the *BeeOS* kernel codebase.
- The macro values are POSIX required and part of the “`fcntl.h`” *libc* header file.

Rule 7.2 (Required) - *A “`u`” or “`U`” suffix shall be applied to all integer constants that are represented in an unsigned type.*

<u>Project deviation</u>

The original kernel does not explicitly append the unsigned suffix to the integer constants and let the compiler do the cast wherever is considered safe. Generally the constants are used as flags, presented in hex form, to be assigned to unsigned types. The unsigned types are then used with binary Boolean and bitwise operators.

Obviously the unsigned suffix for such an constant literal values would be more appropriate because of their usage, but for coherence with the other *BeeOS* projects, i.e. the standard C library, the user space library and the user space programs, the kernel integer constants are left untouched.

The violation could be very easily fixed in the eventuality that is absolutely required.

Note: this deviation is strictly linked with the violations of Rules 10.1, 10.3 and 10.4

Rule 7.3 (Required) - *The lowercase character “l” shall not be used in a literal suffix.*

Compliant.

Rule 7.4 (Required) - *A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char”.*

There was one trivial violation and has been fixed.

```
char *echo_buf;
echo_buf = "\b \b";
```

4.3.8 - Declarations and definitions

Rule 8.1 (Required) - *Types shall be explicitly specified.*

There were four violations within the “`ipc/msg.h`” and “`ipc/msg.c`” sources.

Three violations was due to the not inclusion of ”`time.h`” header and usage of `time_t` type. One violation was due to the `key_t` type missing definition (taken by default as an `int`).

Because were effectively unused (dead code), the non-compliant sources were removed from the project.

Rule 8.2 (Required) - *Function types shall be in prototype form with named parameters.*

There was some violations because of not explicit void parameter type in functions taking no parameters. Easily fixed:

```
int func() → int func(void)
```

Rule 8.3 (Required) - *All declarations of an object or function shall use the same names and type qualifiers.*

There were 16 violations caused by a mismatch in the parameters names in prototypes and definitions of some functions.

Rule 8.4 (Required) - *A compatible declaration shall be visible when an object or function with external linkage is defined.*

There were 70 violations because header files declaring the exported function was not directly included in the file defining the function body.

This practice was not followed because is practically not required. But indeed is safer to do so, the code was thus fixed.

There are also a small number of warning instances caused by the absence of a header file defining the external linkage functions. Where required these were directly declared within the translation unit. Easily fixed as well.

Specific deviations

Two violations remains: `kmain` and `arch_init`, both functions are called by the early startup assembly code. Thus a header file declaration would be superfluous and misleading.

Rule 8.5 (Required) - *An external object or function shall be declared once in one and only one file.*

Some functions or global variables were declared both in the proper header file and in the translation unit were are used.

This is because, probably, the header file were introduced later, during the development.

Rule 8.6 (Required) - *An identifier with external linkage shall have exactly one external definition.*

False positives

Found 65 violations and all are false positives.

The majority were due to their definition within assembly language sources; the other cases were standard C library functions: `strlen, strcpy, memcpy, memset, vsnprintf`.

Rule 8.7 (Advisory) - *Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.*

There were four violations within four syscall handlers that were not inserted into the syscalls array, thus they were dead code. Easily fixed.

False positives

There are five violations caused by C code that is called by external assembly code. The more obvious false positive is the `kmain` function that is invoked by the low level startup routine.

Project deviation

There are some functions that have external linkage and are inserted in the header files even if they are never called by another translation unit. This is voluntary and is intended for possible future usage.

Rule 8.8 (Required) - *The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.*

False positives

The tool signaled two functions: `arch_init` and `kmain`. Both of them are invoked from assembler code thus the violations are taken as false positives.

Rule 8.9 (Advisory) - *An object should be defined at block scope if its identifier only appears in a single function.*

Project deviation

The rule is not strictly followed. Some “long” arrays were declared globally within the translation unit instead of directly within the function, even if its identifier is used only there.

This practice has been followed for two core reasons, one practical and one more bound to code design:

- Is not a good practice do declare huge structures within the stack (especially in embedded systems). Stack clash with some other parts of the program are never easy to detect and are very dangerous. Note this not risk doesn’t exists when the data is declared as `static`.
- Don’t masquerade what the function is supposed to do with huge data declarations.

Rule 8.10 (Required) - *An `inline` function shall be declared with the `static` storage class.*

Compliant.

Rule 8.11 (Advisory) - *When an array with external linkage is declared, its size should be explicitly specified.*

Compliant.

Rule 8.12 (Required) - *Within an enumerator list, the value of an implicitly specified enumeration constant shall be unique.*

Compliant.

Rule 8.13 (Advisory) - *A pointer should point to a `const` qualified type whenever possible.*

Where possible, constant pointers are used to refer to variables that are not modified within the pointer current scope.

Rule 8.14 (Required) - *The `restrict` type qualifier shall not be used.*

Compliant.

4.3.9 - Initialization

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

One fix, the rest are all false positives.

False positives

Three violations are signaled in the x86 architecture specidic input functions (“`io.h`”). The tool doesn’t notice that the variables are initialized via some inline assembly code.

Rule 9.2 (Required) - *The initializer for an aggregate or union shall be enclosed in braces.*

Compliant.

Rule 9.3 (Required) - *Arrays shall not be partially initialized.*

Compliant.

Rule 9.4 (Required) - *An element of an object shall not be initialized more than once.*

Compliant.

Rule 9.5 (Required) - *Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.*

There was a big array, the one holding syscalls handlers, that was using designated initializers without declaring its size explicitly. Easily fixed.

4.3.10 - The essential type model

Compliance to this section requires a lot more expensive code refactory.

The required modifications would be so invasive that the risk to insert bugs within the kernel is higher than the obtained benefit. The code compliance is still possible but requires a time cost beyond the scopes of this code analysis.

As a more personal opinion, in very low level contexts, as in kernel programming, it should be just accepted that C is not a type safe language and that the programmer must know how to use its tools.

Some of the violations were corrected as a side effect of others modifications done during the analysis of the other rules and directives.

Rule 10.1 (Required) - *Operands shall not be of an inappropriate essential type.*

The tool originally notified 264 violations. Of those, 81 are ignored because they are because of flags defined within the standard C library scope, thus not properly part of the project. The others were mitigated to 100 violations as a side effect of code refactory.

Project deviation

Every violation to this rule is due to application of bit-wise operators (bitwise logical or shifts) to literal integer values, that defaults to signed. For example,

```
unsigned int flags = FLAGS;
unsigned int res = flags & 0x1;   /* not compliant */
unsigned int res2 = flags & 0x1U; /* compliant */
```

Compliance to this rule could be trivially obtained by adding the `U` suffix to every literal integer value involved in a logical operation. This modification is not expensive and could be easily applied, but we’ve decided to introduce the above deviation to maintain a consistent coding style with the rest of the project.

For more information refer to Rule 8.2.

Rule 10.2 (Required) - *Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.*

Compliant.

Rule 10.3 (Required) - *The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.*

320 Violations mitigated to 263 via code refactory.

Proper compliance to this rule requires a very expensive intervention.

Project deviation

Almost every violation is due to assignment between unsigned typed variables and integer literal constants. Follows a very trivial example:

```
unsigned int a = 30;  /* non compliant code */
unsigned int a = 30U; /* compliant code */
```

Rule 10.4 (Required) - *Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.*

306 Violations.

Proper compliance to this rule requires a very expensive intervention.

Project deviation

Almost every violation is due to an arithmetic operation performed between unsigned typed variables and integer literal constants. Follows a very trivial example:

```
unsigned int a;
...
if (a < 30) { ... }  /* non compliant code */
if (a < 30U) { ... } /* compliant code */
```

Rule 10.5 (Advisory) - *The value of an expression should not be cast to an inappropriate essential type.*

Compliant.

Rule 10.6 (Required) - *The value of a composite expression shall not be assigned to an object with wider essential type.*

Specific deviation

There are three composite expressions assigned to an object with a wider essential type for context specific requirements. Because truncation is not possible, deviations to this rule can be considered safe in every violation context.

Example of a found violation within the ext2 file-system driver:

```
uint32_t blk;
uint32_t blk  = (dsb.log_block_size == 0) ? 3 : 2; /* not compliant */
```

Compliant version

```
uint32_t blk  = (dsb.log_block_size == 0) ? (uint32_t)3 : (uint32_t)2;
```

The corrections required for compliance are trivial as far as they are ugly. We’ve decided to leave the code *as-is*. In addition is still not very clear why this rule has been marked as required.

Rule 10.7 (Required) - *If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.*

Compliant.

Rule 10.8 (Required) - *The value of a composite expression shall not be cast to a different essential type category or a wider essential type.*

Compliant.

4.3.11 - Pointer type conversions

Low level code, very often, requires to cast pointers to integers, integers to pointers or pointers of one type to pointers of another.

Especially in architecture specific code, if done correctly, the practice can be considered safe. For example, in x86 arch specific code a pointer can be safely casted to and from a `uint32_t` value but not to a `uint16_t` value without potential information lost.

Rule 11.1 (Required) - *Conversions shall not be performed between a pointer to a function and any other type.*

Specific deviation

We ignore all the violations caused by the external macros `SIG_IGN`, `SIG_DFL` and `SIG_ERR`. These macros values are integers casted to `sighandler_t` function pointers and meant to be assigned to the dedicated member of the `siginfo` structure.

These macros with integer values (practically impossible to be assumed by a function pointer) are required by the POSIX standard.

Specific deviation

Within the “`syscall.c`” source file there are 38 implemented syscalls and each one has an associated unique numeric identifier. To invoke a particular kernel syscall the user provides that identifier, eventually with context specific arguments.

To simplify and optimize the selection of the correct syscall handler, all the function pointers are stored within a single array using the identifier as the index. The problem here is that every handling function pointer has a different prototype, thus, to avoid compiler warnings, we’ve decided to store them all in an array of pointers to `void`.

```
static void *syscalls[SYSCALLS_NUM] = {
    [__NR_exit] = sys_exit,
    [__NR_fork] = sys_fork,
    ...
};
```

Specific deviation

Within the *x86* arch specific "`idt.c`" source file there are 49 deviations due to the cast of function pointers to `uint32_t`. This cast can be considered safe because has been performed in an architecture specific context where, the size of a pointer is exactly four bytes.

Additionally, in this particular case there wasn't another other choice, we have to do the cast in order to split the address in two parts (low and high) and place the two chunks in one of the IDT (Interrupt Descriptor Table) entry members.

```
static void idt_entry_init(int i, uint32_t offset, uint16_t selector,
                           uint8_t flags)
{
    ...
    idt_entries[i].offset_lo  = offset & 0x0000FFFF;
    idt_entries[i].offset_hi  = offset >> 16;
    ...
}

idt_entry_init(3, (uint32_t)isr_0, 0x08, 0x8E); /* isr_0 is the handler */
```

Because of their nature all these violations are ignored.

Rule 11.2 (Required) - *Conversions shall not be performed between a pointer to an incomplete type and any other type.*

Compliant.

Rule 11.3 (Required) - *A cast shall not be performed between a pointer to object type and a pointer to a different object type.*

We've received 64 violations to this rule and all of them can be considered safe deviations.

Project deviation

The majority of the violations are due to casts between pointers of different types to easily and efficiently implement some functionality.

One simple example, that covers the majority of the violations, is the macro `struct_ptr`. This macro is widely (and wisely) used to get a pointer to an object from a pointer to a member of the same object.

```
#define struct_ptr(memb_ptr, struct_typ, memb_name) \
   ((struct_typ *)((char *)(memb_ptr)-offsetof(struct_typ,memb_name)))
```

To avoid memory access issues is important that the `struct_typ` is already instanced and properly aligned.

Rule 11.4 (Advisory) - *A conversion should not be performed between a pointer to object and an integer type.*

We've received 89 violations to this rule. 41 instances were resolved by revieweing the code, 32 instances are considered safe because of their architecture specific context, the remaining violations are considered (safe) project deviations.

Project deviation

A pointer to integer and integer to pointer cast is considered safe when the integer is a `uintptr_t` type (indeed that is the purpose of the type).

Once that a pointer has been casted to a `uintptr_t` bitwise operations are allowed.

For example, to align an address up to the next multiple of the size of a `void` pointer.

```
#define ALIGN_UP(val, a) (((val) + ((a) – 1)) & ~((a) - 1))

char *aligned_ptr = ALIGN_UP((uintptr_t)addr, sizeof(void *));
```

Specific deviation

Another deviation comes in the construction of the process stack when it should contain an address. Without recurring to the less manageable double pointers (e.g. `void **`).

```
uintptr_t *sp;
sp[n] = (uintptr_t)sp; /* store the stack pointer address into the stack */
```

Rule 11.5 (Advisory) - *A conversion should not be performed from pointer to `void` into pointer to object.*

Almost every notified violation was due to the `kmalloc` usage. The allocator function returns a pointer to `void` and was usually directly assigned to the destination object pointer. These specific type of violations were solved by casting the returned pointer to the appropriate type before assignment.

Specific deviations

Some specific deviations are required to be compliant with the POSIX standard. As an example, consider the `read` syscall:

```
ssize_t read(int fd, void *buf, size_t count);
```

Internally there should be a point where the implementation casts the `void` pointer to a `uint8_t` (or `unsigned char`) pointer to read the data, octet after octet, from the low level driver.

Rule 11.6 (Required) - *A cast shall not be performed between a pointer to `void` and an arithmetic type.*

We've received 169 violation to this rule.

More than 40 instances were resolved by revieweing the C standard library `offsetof` macro implementation.

Around 100 instances are considered safe due to their architecture specific nature, in such a context casts are inevitable (e.g. to populate the *mmu* page tables)

All the remaining instances are safe project deviations (see Rule 11.4).

Rule 11.7 (Required) - *A cast shall not be performed between pointer to object and a non-integer arithmetic type.*

Compliant.

Rule 11.8 (Required) - *A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.*

There were 5 violations due to cast-away of the `const` qualifier during assignment.

Rule 11.9 (Required) - *The macro `NULL` shall be the only permitted form of integer null pointer constant.*

There were 9 violations due to the direct assignment of the constant value 0 to a pointer type.

4.3.12 - Expressions

Rule 12.1 (Advisory) - *The precedence of operators within expressions should be made explicit.*

According to this rule every composite expression within a conditional statement should be parenthesized to make the operator precedence explicit.

Project deviation

Due to coding style, when the evaluation order is "*not ambiguous*[1]", we introduce a violation.

Example of a non ambiguous expression:

```
res = (a == 0 || b != 0)
```

Example of an ambiguous composite expressions:

```
res = 3 + (1 != 0) ? 1 : 2;
```

With the above expression the 80% of the programmers (we've made a little poll in our office) replies that the composite expression evaluates to 4 (obviously you must don't know that there is a trick...otherwise you have more chances). Actually, the expression evaluates to 1 due to the highest precedence of the operator "+" with respect to the "?".

Parenthesizing, the expression is equivalent to:

1 Subjective. In this case for non-ambiguous we intend cases that are so common that every programmer, worthy of the name, can evaluate correctly.

```
res = (3 + (1 != 0)) ? 1 : 2  → res = 4 ? 1 : 2  → res = 1
```

Obviously our original response followed the wrong precedence path equivalent to:

```
res = 3 + ((1 != 0) ? 1 : 2);
```

Rule 12.2 (Required) - *The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.*

Compliant.

Rule 12.3 (Advisory) - *The comma operator should not be used.*

Every comma operator usage has been removed.

The tool still indicates 8 violations and all are caused by the implementation of the macros operating on the `sigset` type (`sigaddset`, `sigdelset`, ...).

The macros are part of the standard C library thus are ignored.

Rule 12.4 (Advisory) - *Evaluation of constant expressions should not lead to unsigned integer wrap-around.*

Compliant.

4.3.13 - Side effects

Rule 13.1 (Required) - *Initializer lists shall not contain persistent side effects.*

Compliant.

Rule 13.2 (Required) - *The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.*

Compliant.

Rule 13.3 (Advisory) - *A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.*

The tool indicates 15 violations and all are due to expressions like the following:

```
buf[i++] = '/';
```

This is a very widespread, and accepted, way to write C code. If properly used, the violation does not produce any side effect and is quite safe.

<u>Project violation</u>

A project-wide deviation has been introduced due to the previous expression types.

Rule 13.4 (Advisory) - *The result of an assignment operator should not be used.*

The tool indicates 44 violations and all are due to the usage of the result of an assignment in a Boolean expression.

```
if ((n = read_bytes()) < 0) { ... }
```

This is a very widespread, and accepted, way to write C code. If properly used, the violation does not introduce any side effect and is quite safe.

<u>Project violation</u>

A project-wide deviation has been introduced due to the previous expression types.

Rule 13.5 (Required) - *The right hand operand of a logical* `&&` *or* `||` *operator shall not contain persistent side effects.*

Two trivial violations were fixed.

<u>False positive</u>

A violation is still signaled within the "`ext2.c`" source file .

```
if(strlen(name) == dirent->name_len &&
   memcmp(name, dirent->name, dirent->name_len) == 0)
```

PC-Lint reported violation: *"Side effects on right hand of logical operator* &&."

Actually we've not found any valid reason for this warning. There are no side effects related to the `memcmp` call. It works on pointers to constant and it is not implemented as a macro that may alter the "name" pointer value. We'll consider this as a false positive.

Rule 13.6 (Mandatory) - *The operand of the* `sizeof` *operator shall not contain any expression which has potential side effects.*

Compliant.

4.3.14 - Control statement expressions

Rule 14.1 (Required) - *A loop counter shall not have essentially floating type.*

Compliant.

Rule 14.2 (Required) - *A `for` loop shall be well-formed.*

Compliant.

Rule 14.3 (Required) - *Controlling expressions shall not be invariant.*

There was one violation due to an “always false” statement within an if condition.

Rule 14.4 (Required) - *The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.*

Found 44 violations to this rule.

The problem was that we was using integer values in place of Boolean values in flow control expressions, exploiting the fact that in C every non-zero value is considered as a Boolean False.

Every violation has been easily fixed:

```
if (var) { ... }   →   if (var != 0) { ... }
```

4.3.15 - Control flow

Rule 15.1 (Advisory) - *The `goto` statement should not be used.*

We dedicated a slightly longer section to this rule to try to leverage the legendary unconditional hate against the `goto` statement.

As a fact, when used properly, the statement can greatly simplify the implementation and improve the design and efficiency of some, otherwise complicated, operations.

The most widespread usage of the the `goto` statement is to implement a primitive form of exception handling and an easy way to rollback partially initialized objects.

Let there is a piece of code that incrementally initialized an object by allocating resources. In case of an error in the middle of the initialization, i.e. when the object is not fully initialized, we should be able to restore the environment original state.

In place of several “`if...else`” statements containing repeated cleanup code or maintaining several flags to know which parts of the object were already initialized, the `goto` statement greatly simplify the work to be (un)done and produces a cleaner and more readable code.

An example follows.

```
if (init_resource(r1) == FAIL)
    goto e0;
if (init_resource(r2) == FAIL)
    goto e1;
...
```

```
if (init_resource(r10) == FAIL)
    goto e9;
...
    return SUCCESS;

    /* Rollback section */
e9: release_resource(r9);
...
e1: release_resource(r1);
e0: return FAIL;
```

The alternative would be a very deeply nested (and awful) `if...else` “forest” with a lot of repeated code to correctly release the resources.

```
if (init_resource(r1) == SUCCESS) {
    if (init_resource(r2) == SUCCESS) {
        if (init_resource(r3) == SUCCESS) {
            ... /* Very deep nesting */
        } else {
            release_resource(r2);
            release_resource(r1);
        }
    } else {
        release_resource(r1);
}
```

When used wisely, to `goto` statement is thus considered safe and, indeed, gives a clear vision of what the code does.

Project deviation

The described “*rollback-on-exception*” pattern has been used wherever is convenient to do so. In particular the violation has been found within the `execve`, `buddy` and `ext2` implementations.

Rule 15.2 (Required) - *The `goto` statement shall jump to a label declared later in the same function.*

Compliant.

Rule 15.3 (Required) - *Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.*

Compliant.

Rule 15.4 (Advisory) - *There should be no more than one `break` or `goto` statement used to terminate any iteration statement.*

Specific deviations

There are five violations (tty.c, execve.c) because of a loop interrupted in different points using the `break` statement. A modification in such a specific case would be easy but would also result is a less clear code.

Rule 15.5 (Advisory) - *A function should have a single point of exit at the end.*

Project deviation

Wherever a sanity check at the beginning of a non trivial function is not satisfied then the function returns immediately with an error code. With such premature unsatisfied conditions an `if...else` statement would cause the function code to be immediately indented just for precondition checking and that is considered a bad-design choice.

Early returns are generally inserted in syscalls entry points, i.e. where the function arguments values cannot be trusted; *inter*-kernel calls input parameters are generally assumed to be safe and respect the function preconditions.

Example within the `close` syscall implementation

```
int sys_close(int fd)
{
    if (fd < 0)
        return -EBADF;
    ...
    return 0;
}
```

Rule 15.6 (Required) - *The body of an iteration-statement or a selection-statement shall be a compound-statement.*

Skipping single statement braces-encapsulation is a very common ad accepted convention.

Project deviation

We've received 291 violations to this rule. Single statement are generally not encapsulated within braces for a code design choice. This practice is too widespread and affirmed across the project to be changed.

There are some exceptions to the following practice, and are generally guided by good sense and code readability. Follows some examples.

The rule is followed when the nesting level is greater than one

```
if(condition1) {   /* This brace can be potentially omitted */
    if (confition2)
        a = b;
}
```

Or to avoid one of the most infamous source of errors in C programming: the *else-ownership-ambiguity*. Because the language ignores indentation without braces, there is no way of separating this:

```
if(one)
    if(two)
        foo();
    else   /* This else is associated with the second if */
        bar();
```

From this (misleading) version of the very same code snip:

```
if(one)
```

```
        if(two)
            foo();
    else       /* This else statement is associated with the second if */
        bar();
```

The else statement is always associated to the closer if statement.

To avoid this error prone language details, we prefer to use braces in such cases.

Rule 15.7 (Required) - *All `if...else if` constructs shall be terminated with an `else` statement.*

<u>Specific deviation</u>

There is one case, with the SLAB allocator (`slab.c`) implementation where there is an "`if...else if`" statement without a default `else`. In this case, the `else` clause would be empty (required by MISRA but not accepted by the Author).

4.3.16 - Switch statements

Rule 16.1 (Required) - *All `switch` statements shall be well-formed.*

Compliant.

Rule 16.2 (Required) - *A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.*

Compliant.

Rule 16.3 (Required) - *An unconditional `break` statement shall terminate every `switch` clause.*

There were some violations were `switch` statements doesn't have the `break` to use a fall-through approach share some code. The required change to attain compliance was minimal so we've adjusted it.

Rule 16.4 (Required) - *Every `switch` statement shall have a `default` label.*

In one switch statement the `default` label was (technically superflous but) missing.

Fixed for compliance.

Rule 16.5 (Required) - *A `default` label shall appear as either the first or the last switch label of a `switch` statement.*

Compliant.

Rule 16.6 (Required) - *Every `switch` statement shall have at least two `switch` clauses*

A trivial switch was present and has been removed.

Rule 16.7 (Required) - *A `switch` expression shall not have essentially Boolean type*.

Compliant.

4.3.17 - Functions

Rule 17.1 (Required) - *The features of "`<stdarg.h>`" shall not be used.*

<u>Specific deviations</u>

There were two deviations, one within the `panic` function and one within the `kprintf`.

Both the functions use variable arguments to implement format strings processing and thus to dynamically print additional information during logging.

Rule 17.2 (Required) - *Functions shall not call themselves, either directly or indirectly.*

Is a well known fact that recursion is considered a bad practice in embedded applications (generally to avoid stack overruns over the data section). As such, it has been avoided with just two exceptions:

<u>Specific deviation</u>

There is an indirect recursion path within the SLAB allocator implementation:

1. The `kmalloc` uses the SLAB allocator.
2. The SLAB allocator, `slab_cache_alloc`, uses `kmalloc` to allocate some control structures.

The recursion happens only when the SLAB allocator context has the flag `SLAB_EMBED_BUFCTL` turned OFF.

The SLAB allocator uses the `kmalloc` using a SLAB allocator context that has the `SLAB_EMBED_BUFCTL` turned ON.

Because of the above conditions the recursion is immediatelly stopped, giving a maximum nesting level of 2:

```
kmalloc → slab_cache_alloc → kmalloc → slab_cache_alloc → return
```

<u>Specific deviation</u>

Similar to the previous one, but with respect to the `kfree` and the `slab_cache_free` functions.

Rule 17.3 (Mandatory) - *A function shall not be declared implicitly.*

Compliant. The rule check is enforced by the compiler.

Rule 17.4 (Mandatory) - *All exit paths from a function with non-void return type shall have an explicit return statement with an expression.*

Compliant.

Compliance is additionally enforced by the compiler.

Rule 17.5 (Advisory) - *The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of element.*

Compliant.

Rule 17.6 (Mandatory) - *The declaration of an array parameter shall not contain the `static` keyword between the [ ].*

Compliant.

Rule 17.7 (Required) - *The value returned by a function having non-`void` return type shall be used.*

There were 50 violations and all were addressed.

In the cases where we don't require the return value and it can be safely ignored the function call has been prefixed with a cast to `void` type (explicit return type cast).

For example, to ignore the `read` return value explicitly:

```
(void)read(fd, buf, 3);
```

Rule 17.8 (Advisory) - *A function parameter should not be modified.*

Project deviation

Where is conventient to do so, e.g. counters that should be decremented or pointers that are accessed sequentially with the same step of the declared type, the function argument value is modified and thus a deviation is introduced.

4.3.18 - Pointers and arrays

Rule 18.1 (Required) - *A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.*

Three violations were found. One has been fixed and the other two are false positives.

False Positive

The tool is unable to infer that the `panic` function never returns. Panic, is the function we call if we detect an out of bounds access tentative.

```
if (index >= ARRAY_SIZE)
    panic(“out of bounds access detected”); /* Never returns */
arr[index] = val; /* Safe assignment */
```

False Positive

The tool wasn’t able to detect that an index is indirectly checked

```
for (fd0 = 0; fd0 < OPEN_MAX; fd0++)
    ...
for (fd1 = fd0 + 1; fd1 < OPEN_MAX; fd1++)
    ...
if (fd1 >= OPEN_MAX)
    return -EMFILE;
current_task->fds[fd0].fil = file0; /* Violation indication here */
```

Because `fd1` is always greater than `fd0`, if `fd1 < OPEN_MAX` then there is no need to check `fd0`.

Rule 18.2 (Required) - *Subtraction between pointers shall only be applied to pointers that address elements of the same array.*

Compliant.

Note: in some contexts, e.g. paging code, we consider the whole process address space as a single array of bytes, the process memory array.

Rule 18.3 (Required) - *The relational operators* >, >=, < *and* <= *shall not be applied to objects of pointer type except where they point into the same object.*

Refer to MISRA C 2012 Rule 18.2.

Rule 18.4 (Advisory) - *The* +, -, += *and* -= *operators should not be applied to an expression of pointer type.*

Project violation

Pointer arithmetic has been used extensively in the project to easily move a pointer into an array of objects and to compute the index of an array element from its address.

Example:

```
struct mystryct ar[10];
struct mystruct *p = get_element(ar);
size_t index = p - ar;  /* get p index */
```

Rule 18.5 (Advisory) - *Declarations should contain no more than two levels of pointer nesting.*

Compliant.

Rule 18.6 (Required) - *The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.*

Compliant.

Rule 18.7 (Required) - *Flexible array members shall not be declared.*

Compliant.

Rule 18.8 (Required) - *Variable-length array types shall not be used.*

Compliant.

4.3.19 - Overlapping storage

We've found no violations to this section. The rules are enumerated for completeness.

Rule 19.1 (Mandatory) - *An object shall not be assigned or copied to an overlapping object.*

Rule 19.2 (Advisory) - *The union keyword should not be used.*

4.3.20 - Preprocessing directives

We've found no violations to this section. The rules are enumerated for completeness.

Rule 20.1 (Advisory) - `#include` *directives should only be preceded by preprocessor directives or comments.*

Rule 20.2 (Required) - *The* `'`, `"` *or* `\` *characters and the* `/*` *or* `//` *character sequences shall not occur in a header file name.*

Rule 20.3 (Required) - *The `#include` directive shall be followed by either a <filename> or "filename" sequence.*

Rule 20.4 (Required) - *A macro shall not be defined with the same name as a keyword.*

Rule 20.5 (Advisory) - *`#undef` should not be used.*

Rule 20.6 (Required) - *Tokens that look like a preprocessing directive shall not occur within a macro argument.*

Rule 20.7 (Required) - *Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.*

Rule 20.8 (Required) - *The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1*

Rule 20.9 (Required) - *All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation*

Rule 20.10 (Advisory) - *The # and ## preprocessor operators should not be used*

Rule 20.11 (Required) - *A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.*

Rule 20.12 (Required) - *A macro parameter used as an operand to the `#` or `##` operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.*

Rule 20.13 (Required) - *A line whose first token is # shall be a valid preprocessing directive.*

Rule 20.14 (Required) - *All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.*

4.3.21 - Standard libraries

Almost every rule listed by this section is more dedicated to user-space applications.

The kernel usage of the standard library functions and macros is limited to the ones contained within the “`string.h`”, “`fcntl.h`”, “`ctype.h`”, “`stdint.h`” header files and to a couple of structures within the forbidden “`time.h`” and “`signal.h`”

Rule 21.1 (Required) - *`#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.*

Originally, there were plenty violations to this rule because every header file *include guard* was starting with an underscore and that is considered a reserved macro name. Easily fixed.

Rule 21.2 (Required) - *A reserved identifier or macro name shall not be declared.*

Specific Deviation

The `ctime` member of the `inode` structure has signaled as a reserved identifier.

We’re not strictly required to use such an identifier since is member of a structure not directly influencing the POSIX interface but we’ve decided to introduce a violation to use the “*traditional*” `inode` creation time used by UNIX like systems.

Rule 21.3 (Required) - *The memory allocation and deallocation functions of <`stdlib.h`> shall not be used.*

Compliant.

Rule 21.4 (Required) - *The standard header file <`setjmp.h`> shall not be used.*

Compliant.

Rule 21.5 (Required) - *The standard header file <`signal.h`> shall not be used.*

The project contains two deviations due to required compliance to the POSIX standard

Specific deviation

The header file has been included by “`task.h`” header file to peek the definition of the `sigset_t` type. The definition is used within the `task` structure.

```
struct task {
    ...
    sigset_t sigpend;
    sigset_t sigmask;
```

```
        ...
    };
```

<u>Specific deviation</u>

The header file has been included by “sys.h” header file to peek the definition of the `sigaction` type. The definition is used by the `sigaction` syscall implementation.

```
    int sys_sigaction(int sig, const struct sigaction *act,
                      struct sigaction *oact);
```

To be compliant, the kernel would have to define its own `sigaction` and `sigset_t` types, but doing so can introduce inconsistencies with the versions provided by the standard C library.

Rule 21.6 (Required) - *The Standard Library input/output functions shall not be used.*

<u>Specific deviation</u>

The kernel function `kvprintf` internally uses the `vsnprintf` to print the format string within a character array before send the result to the kernel log output. The `vsnprintf` is considered secure because of boundary checking.

Rule 21.7 (Required) - *The `atof`, `atoi`, `atol` and `atoll` functions of <`stdlib.h`> shall not be used*

Compliant.

Rule 21.8 (Required) - *The library functions `abort`, `exit`, `getenv` and `system` of <`stdlib.h`> shall not be used*

Compliant.

We are the providers of the the `exit` function backend as a syscall (`sys_exit`).

Rule 21.9 (Required) - *The library functions `bsearch` and `qsort` of <`stdlib.h`> shall not be used*

Compliant.

Rule 21.10 (Required) - *The Standard Library time and date functions shall not be used.*

<u>Specific deviation</u>

The standard header file “`time.h`” has been included by “`sys.h`” to peek the definition of the `timespec` structure. The definition is used by the `nanosleep` syscall implementation.

```
    int sys_nanosleep(const struct timespec *req, struct timespec *rem);
```

The tool warning is propagated to every file that directly or indirectly includes `sys.h`.

To be compliant with MISRA C, the kernel would have to define its own `timespec` type, but doing so can introduce inconsistencies with the version provided by the standard C library.

Rule 21.11 (Required) - *The standard header file `<tgmath.h>` shall not be used.*

Compliant.

Rule 21.12 (Advisory) - *The exception handling features of `<fenv.h>` should not be used.*

Compliant.

4.3.22 - Resources

We’ve found no violations to this section. Thus we limit to enumerate the rules as a reference.

Rule 22.1 (Required) - *All resources obtained dynamically by means of Standard Library functions shall be explicitly released.*

Rule 22.2 (Mandatory) - *A block of memory shall only be freed if it was allocated by means of a Standard Library function.*

Rule 22.3 (Required) - *The same file shall not be open for read and write access at the same time on different streams.*

Rule 22.4 (Mandatory) - *There shall be no attempt to write to a stream which has been opened as read-only.*

Rule 22.5 (Mandatory) - *A pointer to a `FILE` object shall not be dereferenced.*

Rule 22.6 (Mandatory) - *The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.*

4.4 Conformance Matrix

Static Analysis Tools: *Understand v4.0, PC-Lint*

Complier*: GNU gcc 7.3.0.*

First pass violations: violations found in the original codebase (v0.1.1).

Final pass violations: violations found in the final codebase (v.0.2.0).

Final pass ignored: false positives.

	Type	**First Pass Violations**	**Final Pass Violations**	**Final Pass Ignored**	**Check Method**
Directive 1.1	Required	0	0	0	PC-Lint
Directive 2.1	Required	0	0	0	Compiler / PC-Lint
Directive 3.1	Required	0	0	0	PC-Lint
Directive 4.1	Required	0	0	0	PC-Lint
Directive 4.2	Advisory	0	0	0	PC-Lint
Directive 4.3	Required	12	0	0	Understand
Directive 4.4	Advisory	0	0	0	Understand
Directive 4.5	Advisory	1	1	0	Understand
Directive 4.6	Advisory	406	405	0	Understand
Directive 4.7	Required	43	0	0	PC-Lint
Directive 4.8	Advisory	85	71	0	Understand
Directive 4.9	Advisory	0	0	0	PC-Lint
Directive 4.10	Required	0	0	0	Understand
Directive 4.11	Required	0	0	0	PC-Lint
Directive 4.12	Required	N	0	0	Understand
Directive 4.13	Advisory	0	0	0	PC-Lint
Rule 1.1	Required	0	0	0	PC-Lint
Rule 1.2	Advisory	27	0	27	PC-Lint
Rule 1.3	Required	0	0	0	PC-Lint
Rule 2.1	Required	0	0	0	Understand
Rule 2.2	Required	38	0	38	PC-Lint
Rule 2.3	Advisory	1	0	1	Understand
Rule 2.4	Advisory	1	0	1	Understand
Rule 2.5	Advisory	63	0	32	Understand
Rule 2.6	Advisory	0	0	0	PC-Lint
Rule 2.7	Advisory	14	0	11	Understand
Rule 3.1	Required	114	0	113	Understand

Rule 3.2	Required	0	0	0	Understand
Rule 4.1	Required	0	0	0	Understand
Rule 4.2	Required	0	0	0	Understand
Rule 5.1	Required	0	0	0	Understand
Rule 5.2	Required	0	0	0	Understand
Rule 5.3	Required	6	0	0	Understand
Rule 5.4	Required	0	0	0	Understand
Rule 5.5	Required	2	0	0	Understand
Rule 5.6	Required	0	0	0	Understand
Rule 5.7	Required	4	0	0	Understand
Rule 5.8	Required	0	0	0	Understand
Rule 5.9	Advisory	2	0	0	Understand
Rule 6.1	Required	0	0	0	Understand
Rule 6.2	Required	0	0	0	Understand
Rule 7.1	Required	49	0	39	PC-Lint
Rule 7.2	Required	31	0	31	Understand
Rule 7.3	Required	0	0	0	Understand
Rule 7.4	Required	1	0	0	PC-Lint
Rule 8.1	Required	4	0	0	PC-Lint
Rule 8.2	Required	9	0	0	Understand
Rule 8.3	Required	16	0	0	Understand
Rule 8.4	Required	79	2	0	Understand
Rule 8.5	Required	7	0	0	Understand
Rule 8.6	Required	65	0	65	Understand
Rule 8.7	Advisory	25	17	5	Understand
Rule 8.8	Required	4	0	2	Understand
Rule 8.9	Required	19	15	0	Understand
Rule 8.10	Required	0	0	0	Understand
Rule 8.11	Advisory	0	0	0	Understand
Rule 8.12	Required	0	0	0	Understand
Rule 8.13	Advisory	55	17	0	PC-Lint
Rule 8.14	Required	0	0	0	Understand
Rule 9.1	Mandatory	7	0	6	PC-Lint
Rule 9.2	Required	0	0	0	PC-Lint
Rule 9.3	Required	0	0	0	PC-Lint
Rule 9.4	Required	0	0	0	PC-Lint
Rule 9.5	Required	32	0	0	PC-Lint
Rule 10.1	Required	264	99	81	PC-Lint
Rule 10.2	Required	0	0	0	PC-Lint
Rule 10.3	Required	320	260	0	PC-Lint

Rule 10.4	Required	306	306	0	PC-Lint
Rule 10.5	Advisory	0	0	0	PC-Lint
Rule 10.6	Required	10	3	0	PC-Lint
Rule 10.7	Required	35	0	0	PC-Lint
Rule 10.8	Required	16	0	0	PC-Lint
Rule 11.1	Required	90	0	90	PC-Lint
Rule 11.2	Required	0	0	0	PC-Lint
Rule 11.3	Required	64	62	0	PC-Lint
Rule 11.4	Advisory	89	16	32	PC-Lint
Rule 11.5	Advisory	44	41	0	PC-Lint
Rule 11.6	Required	169	28	101	PC-Lint
Rule 11.7	Required	0	0	0	PC-Lint
Rule 11.8	Required	5	0	0	PC-Lint
Rule 11.9	Required	8	0	0	PC-Lint
Rule 12.1	Advisory	166	157	0	PC-Lint
Rule 12.2	Required	2	0	0	PC-Lint
Rule 12.3	Advisory	18	0	8	PC-Lint
Rule 12.4	Advisory	0	0	0	PC-Lint
Rule 13.1	Required	0	0	0	PC-Lint
Rule 13.2	Required	0	0	0	PC-Lint
Rule 13.3	Advisory	15	9	0	PC-Lint
Rule 13.4	Advisory	42	15	0	PC-Lint
Rule 13.5	Required	3	0	1	PC-Lint
Rule 13.6	Mandatory	0	0	0	PC-Lint
Rule 14.1	Required	0	0	0	PC-Lint
Rule 14.2	Required	0	0	0	PC-Lint
Rule 14.3	Required	1	0	0	PC-Lint
Rule 14.4	Required	44	0	0	PC-Lint
Rule 15.1	Advisory	8	8	0	Understand
Rule 15.2	Required	0	0	0	Understand
Rule 15.3	Required	0	0	0	Understand
Rule 15.4	Advisory	6	0	5	Understand
Rule 15.5	Advisory	72	69	0	Understand
Rule 15.6	Required	291	252	0	Understand
Rule 15.7	Required	1	1	0	Understand
Rule 16.1	Required	0	0	0	Manual
Rule 16.2	Required	0	0	0	Understand
Rule 16.3	Required	5	0	0	Understand
Rule 16.4	Required	1	0	0	Understand
Rule 16.5	Required	0	0	0	Understand

Rule 16.6	Required	1	0	0	Understand
Rule 16.7	Required	0	0	0	Manual
Rule 17.1	Required	3	2	0	Understand
Rule 17.2	Required	9	9	0	Understand
Rule 17.3	Mandatory	0	0	0	Understand
Rule 17.4	Mandatory	0	0	0	Compiler / PC-Lint
Rule 17.5	Advisory	0	0	0	PC-Lint
Rule 17.6	Mandatory	0	0	0	Understand
Rule 17.7	Required	50	0	0	Understand
Rule 17.8	Advisory	49	23	0	Understand
Rule 18.1	Required	3	0	2	PC-Lint
Rule 18.2	Required	0	0	0	PC-Lint
Rule 18.3	Required	0	0	0	PC-Lint
Rule 18.4	Advisory	30	28	0	PC-Lint
Rule 18.5	Advisory	0	0	0	PC-Lint
Rule 18.6	Required	0	0	0	PC-Lint
Rule 18.7	Required	0	0	0	PC-Lint
Rule 18.8	Required	0	0	0	PC-Lint
Rule 19.1	Mandatory	0	0	0	PC-Lint
Rule 19.2	Advisory	0	0	0	Understand
Rule 20.1	Advisory	0	0	0	Understand
Rule 20.2	Required	0	0	0	Understand
Rule 20.3	Required	0	0	0	Understand
Rule 20.4	Required	0	0	0	Understand
Rule 20.5	Advisory	0	0	0	Understand
Rule 20.6	Required	0	0	0	Understand
Rule 20.7	Required	0	0	0	PC-Lint
Rule 20.8	Required	0	0	0	PC-Lint
Rule 20.9	Required	0	0	0	PC-Lint
Rule 20.10	Advisory	0	0	0	Understand
Rule 20.11	Required	0	0	0	Understand
Rule 20.12	Required	0	0	0	PC-Lint
Rule 20.13	Required	0	0	0	Understand
Rule 20.14	Required	0	0	0	Understand
Rule 21.1	Required	120	0	0	Understand
Rule 21.2	Required	2	2	0	Understand
Rule 21.3	Required	0	0	0	Understand
Rule 21.4	Required	0	0	0	Understand
Rule 21.5	Required	2	0	3	Understand
Rule 21.6	Required	0	0	0	Understand

Rule 21.7	Required	0	0	0	Understand
Rule 21.8	Required	0	0	0	Understand
Rule 21.9	Required	0	0	0	Understand
Rule 21.10	Required	43	0	43	Understand
Rule 21.11	Required	0	0	0	Understand
Rule 21.12	Advisory	0	0	0	Understand
Rule 22.1	Required	0	0	0	PC-Lint
Rule 22.2	Mandatory	0	0	0	PC-Lint
Rule 22.3	Required	0	0	0	PC-Lint
Rule 22.4	Mandatory	0	0	0	PC-Lint
Rule 22.5	Mandatory	0	0	0	PC-Lint
Rule 22.6	Mandatory	0	0	0	PC-Lint

4.4.1 - Recap

Section	**Subject**	**First Pass Violations**	**Final Pass Violations**	**Final Pass Ignored**
Directive 1	Implementation	0	0	0
Directive 2	Compilation and build	0	0	0
Directive 3	Requirements traceability	0	0	0
Directive 4	Code design	547	477	0
Rule 1	Standard C environment	27	0	27
Rule 2	Unused code	117	0	82
Rule 3	Comments	114	0	113
Rule 4	Character sets and lexical conventions	0	0	0
Rule 5	Identifiers	14	0	0
Rule 6	Types	0	0	0
Rule 7	Literals and constants	81	0	70
Rule 8	Declarations and definitions	283	47	72
Rule 9	Initialization	39	0	6
Rule 10	Essential type model	951	667	81
Rule 11	Pointer type conversions	467	147	223
Rule 12	Expressions	186	157	0
Rule 13	Side Effects	60	24	1
Rule 14	Control Statement Expressions	45	0	0
Rule 15	Control Flow	378	330	5
Rule 16	Switch statements	7	0	0
Rule 17	Functions	111	34	0
Rule 18	Pointers and arrays	33	28	2

Rule 19	Overlapping storage	0	0	0
Rule 20	Preprocessing directives	0	0	0
Rule 21	Standard library	167	2	46
Rule 22	Resources	0	0	0
	TOTAL	**3627**	**1912**	**728**

5 MISRA C: 2004

5.1 Brief

In 2004, "*Guidelines for the use of the C language in critical systems*", or *MISRA-C 2004* was produced. The standard, compared to its predecessor (MISRA-C 1998), contains many substantial changes to the guidelines, including a complete renumbering of the rules.

MISRA C 2004 contains **142 rules**, of which 122 required and 20 advisory. They are divided into 21 topical categories.

Differently from the detailed analysis conduced for the MISRA C 2012 rules set, instead to enumerate all the rules and the details, our attention will be mainly focused on the non-conformances and the motivations behind the violations.

Wherever there is a rule deviation that is compatible with a MISRA 2012 directive or rule a reference to that is inserted to avoid duplication.

Because only only the *Understand* tool will be used for this analysis, some rules are not automatically checked at all.

5.2 Rules

5.2.1 - Identifiers

Rule 5.6 (Advisory) - *No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.*

Refer to MISRA 2004 Rule 5.7.

Rule 5.7 (Advisory) - *No identifier name should be reused.*

This rule mandates the usage of a different name <u>for every</u> variable within the project context.

<u>Project deviation</u>

This rule has not been followed because it requires a huge, and - in our opinion - senseless, code refactory.

5.2.2 - Types

Rule 6.3 (Advisory) - *Typedefs that indicate size and signedness should be used in place of the basic numerical types.*

Refer to MISRA 2012 Directive 4.6.

5.2.3 - Declarations and definitions

Rule 8.5 (Required) - *There shall be no definitions of objects or functions in a header file.*

For some tools this rule indirectly forbids usage of `inline` functions.

Project deviation

The kernel contains some `inline` functions where has been considered useful to do so.

Wherever a very short code sequence can be implemented as a macro, generally when a return value is not required, macro usage has been preferred.

Obviously, compliance can be obtained by avoiding `inline` functions (that indeed are not even included by the C89 version of the standard) but for very short code snip would be an overkill to define it as a stand-alone function.

Rule 8.7 (Required) - *Objects shall be defined at block scope if they are only accessed from within a single function.*

Refer to MISRA 2012 Rule 8.9.

Rule 8.10 (Required) - *All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.*

False positives

The *Understand* tool indicates violations for functions that are referenced from the assembly language code.

Rule 8.11 (Required) - *The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.*

Refer to MISRA 2004 Rule 8.10.

5.2.4 - Arithmetic type conversions

Rule 10.6 (Required) - *A “`U`” suffix shall be applied to all constants of unsigned type.*

Refer to MISRA 2012 Rule 7.2.

5.2.5 - Control flow

Rule 14.4 (Required) - *The `goto` statement shall not be used.*

Refer to MISRA 2012 Rule 15.1.

Rule 14.6 (Required) - *For any iteration statement there shall be at most one `break` statement used for loop termination.*

Refer to MISRA 2012 Rule 15.4.

Rule 14.7 (Required) - *A function shall have a single point of exit at the end of the function.*

Refer to MISRA 2012 Rule 15.5.

Rule 14.8 (Required) - *The statement forming the body of a `switch`, `while`, `do...while` or `for` statement shall be a compound statement.*

Refer to MISRA 2012 Rule 15.6.

Rule 14.9 (Required) - *An `if` (condition) construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.*

Refer to MISRA 2012 Rule 15.6.

Rule 14.10 (Required) - *All `if ... else if` constructs shall be terminated with an `else` clause.*

Refer to MISRA 2012 Rule 15.7.

5.2.6 - Functions

Rule 16.1 (Required) - *Functions shall not be defined with variable numbers of arguments.*

Specific deviations

There are two violations and both are for the same motivation. The non-compliant functions purpose is activity logging by using the same, well known, `printf` format strings interface.

Non-compliant functions prototypes:

```
void kprintf(const char *fmt, ...);

void panic(const char *fmt, ...);
```

Rule 16.2 (Required) - Functions shall not call themselves, either directly or indirectly.

Refer to MISRA 2012 Rule 17.2.

Rule 16.9 (Required) - *A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.*

In C wherever a function identifier is used in an expression and is not the operand of the unary `&` operator, it is implicitly converted to a pointer.

Project deviation

Function identifier implicit pointer conversion is so widely accepted and radicated within the kernel that is not worth to be changed.

5.2.7 - Preprocessing directives

Rule 19.7 (Advisory) - *A function should be used in preference to a function-like macro.*

Project deviation

Function-like macros are used anywere is convenient to do so. When a value should be returned `inline` function are preferred.

Rule 19.10 (Required) - *In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.*

False positives

Three violations were found, all with the following form:

```
#define list_container(link, type, member) \
        ((type *) ((char *)(link) - offsetof(type, member)))
```

The violation is caused by the “`type`” parameter. This parameter cannot be parenthesized within the macro body for language syntax constraints.

5.2.8 - Standard libraries

Rule 20.4 (Required) - *Dynamic heap memory allocation shall not be used.*

Refer to MISRA 2012 Directive 4.12.

Rule 20.6 (Required) - *The macro `offsetof`, in library "`<stddef.h>`", shall not be used.*

<u>Specific deviation</u>

The macro is used to implement a core function within the *kernel-lists* implementation.

A list node is embedded within the linked structure and the `offsetof` macro is used to get the container from the embedded list link.

For more details refer to MISRA 2012 Rule 11.3.

Rule 20.8 (Required) - *The signal handling facilities of "`<signal.h>`" shall not be used.*

Refer to MISRA 2012 Rule 21.5.

Rule 20.9 (Required) - *The input output library "`<stdio.h>`" shall not be used in production code.*

Refer to MISRA 2012 Rule 21.6.

Rule 20.12 (Required) - *The time handling functions of library "`<time.h>`" shall not be used.*

Refer to MISRA 2012 Rule 21.10.

5.3 Conformance Matrix

Static Analysis Tools: *Understand v4.0*

Complier*: GNU gcc 7.3.0*

First pass violations: violations found in the POST MISRA-C 2012 codebase.

Final pass violations: violations found in the final codebase (v0.2.0).

Final pass ignored: false positives.

	Type	**First Pass Violations**	**Final Pass Violations**	**Final Pass Ignored**	**Check Method**
Rule 2.1	Required	12	0	0	Understand
Rule 2.2	Required	66	0	0	Understand
Rule 2.3	Required	0	0	0	Understand
Rule 2.4	Advisory	2	0	0	Understand
Rule 4.1	Required	0	0	0	Understand
Rule 4.2	Required	0	0	0	Understand
Rule 5.1	Required	0	0	0	Understand
Rule 5.2	Required	6	0	0	Understand
Rule 5.3	Required	0	0	0	Understand
Rule 5.4	Required	45	0	0	Understand
Rule 5.5	Advisory	2	0	0	Understand
Rule 5.6	Advisory	146	126	0	Understand
Rule 5.7	Advisory	897	897	0	Understand
Rule 6.3	Required	406	405	0	Understand
Rule 6.4	Required	0	0	0	Understand
Rule 6.5	Required	0	0	0	Understand
Rule 7.1	Required	0	0	0	Understand
Rule 8.3	Required	24	0	0	Understand
Rule 8.5	Required	49	45	0	Understand
Rule 8.6	Required	1	0	0	Understand
Rule 8.7	Required	19	15	0	Understand
Rule 8.8	Required	16	0	0	Understand
Rule 8.9	Required	0	0	0	Understand
Rule 8.10	Required	37	0	4	Understand
Rule 8.11	Required	35	0	2	Understand
Rule 8.12	Required	0	0	0	Understand
Rule 9.3	Required	0	0	0	Understand

Rule 10.6	Required	32	31	0	Understand
Rule 12.6	Advisory	0	0	0	Understand
Rule 12.12	Required	0	0	0	Understand
Rule 13.3	Required	0	0	0	Understand
Rule 13.6	Required	0	0	0	Understand
Rule 14.1	Required	0	0	0	Understand
Rule 14.3	Required	0	0	0	Understand
Rule 14.4	Required	8	8	0	Understand
Rule 14.5	Required	8	1	0	Understand
Rule 14.6	Required	6	5	0	Understand
Rule 14.7	Required	71	68	0	Understand
Rule 14.8	Required	23	20	0	Understand
Rule 14.9	Required	267	265	0	Understand
Rule 14.10	Required	1	1	0	Understand
Rule 15.1	Required	0	0	0	Understand
Rule 15.2	Required	5	0	0	Understand
Rule 15.3	Required	1	0	0	Understand
Rule 15.5	Required	1	0	0	Understand
Rule 16.1	Required	4	2	2	Understand
Rule 16.2	Required	9	9	0	Understand
Rule 16.3	Required	0	0	0	Understand
Rule 16.4	Required	0	0	0	Understand
Rule 16.5	Required	8	0	0	Understand
Rule 16.9	Required	61	61	0	Understand
Rule 17.5	Advisory	0	0	0	Understand
Rule 18.4	Required	0	0	0	Understand
Rule 19.1	Advisory	0	0	0	Understand
Rule 19.2	Advisory	0	0	0	Understand
Rule 19.3	Required	0	0	0	Understand
Rule 19.5	Required	0	0	0	Understand
Rule 19.6	Required	0	0	0	Understand
Rule 19.7	Advisory	35	35	0	Understand
Rule 19.9	Required	0	0	0	Understand
Rule 19.10	Required	5	0	3	Understand
Rule 19.11	Required	1	0	0	Understand
Rule 19.12	Required	0	0	0	Understand
Rule 19.13	Advisory	0	0	0	Understand
Rule 19.14	Required	0	0	0	Understand
Rule 19.15	Required	0	0	0	Understand
Rule 19.17	Required	0	0	0	Understand

Rule 20.1	Required	35	0	0	Understand
Rule 20.2	Required	35	0	0	Understand
Rule 20.4	Required	0	0	0	Understand
Rule 20.5	Required	0	0	0	Understand
Rule 20.6	Required	40	40	0	Understand
Rule 20.7	Required	0	0	0	Understand
Rule 20.8	Required	2	2	0	Understand
Rule 20.9	Required	4	1	0	Understand
Rule 20.10	Required	0	0	0	Understand
Rule 20.11	Required	0	0	0	Understand
Rule 20.12	Required	1	1	0	Understand
Rule 21.1	Required	0	0	0	Understand

5.3.1 - Recap

NA: **N**ot **A**utomatically checked via a static analysis tool. Best effort has been followed to avoid violations.

Section	Subject	First Pass Violations	Final Pass Violations	Final Pass Ignored
Rule 1	Environment	NA	NA	NA
Rule 2	Language extensions	80	0	0
Rule 3	Documentation	NA	NA	NA
Rule 4	Character sets	0	0	0
Rule 5	Identifiers	1096	1023	0
Rule 6	Types	406	405	0
Rule 7	Constants	0	0	0
Rule 8	Declarations and definitions	181	60	6
Rule 9	Initialization	0	0	0
Rule 10	Arithmetic type conversions	32	31	0
Rule 11	Pointer type conversions	NA	NA	NA
Rule 12	Expressions	0	0	0
Rule 13	Control statement expressions	0	0	0
Rule 14	Control flow	384	368	0
Rule 15	Switch statements	7	0	0
Rule 16	Functions	82	72	2
Rule 17	Pointers and arrays	0	0	0

Rule 18	Structures and unions	0	0	0
Rule 19	Preprocessing directives	41	35	3
Rule 20	Standard libraries	117	44	0
Rule 21	Run-time failures	0	0	0
	TOTAL	**2426**	**2038**	**11**

6 SEI CERT C

6.1.1 - Brief

The CERT C rules were developed by the SEI CERT Coordination Center. The CERT/CC is the coordination center of the Computer Emergency Response Team (CERT) for the Software Engineering Institute (SEI) hosted by the Carnegie Mellon University.

This standard provides rules for secure coding in the C programming language. The goal of these **rules** and **recommendations** is to develop safe, reliable, and secure systems, for example by eliminating undefined behaviors that can lead to exploitable vulnerabilities.

As for MISRA, conformance to the coding rules defined by the standard are a necessary (but not sufficient) to ensure the safety, reliability, and security of a software system.

CERT's coding standards are being widely adopted by industry.

6.1.2 - Static Analysis

It is possible to check that C source code complies with CERT C rules and recommendation by means of inspection alone. However, this is likely to be extremely time-consuming and error prone.

Any realistic process for checking code against CERT C will therefore involve the use of at least one **static analysis tool**.

CERT C rules are verified through the CERT C ***Rosecheckers*** parser supported by the ROSE analyzer.

The tool is a bit tricky to install to the point that the Authors decided to make it available pre-installed in a Virtual Machine (https://sourceforge.net/projects/rosecheckers/)[2].

Fortunately a quicker way to work exists thanks to a Docker user that decided to share with the community an image with ROSE and the *Rosecheckers* ready for use. The solution is similar to the VM but a bit lighter and faster.

Once you've installed Docker (e.g. a Debian package exists) you just have to start the container with the following command:

```
$ docker run -ti --rm -v /hostsrc:/src kontotto/rosecheckers /bin/bash

  "-ti" : redirect tty io to the host
  "--rm" : remove the docker container on exit (otherwise on exit is just stopped)
  "-v /hostsrcpath:/src" : map the absolute host src path to the container /src dir.
  "knontotto/rosecheckrs" : the docker-hub user and its docker image
  "/bin/bash" : spawn a shell within the running container
```

2 I would find less invasive if they've made a linux package installable through the common package management tools (e.g. apt). Probably it would required some maintenance for cross version portability.

Now you can enter the `/src` directory within the container and issue the following command

```
make CC=rosecheckers
```

Here is a breakdown, from the SEI website, of how thoroughly *Rosecheckers* enforces the C Secure Coding Rules and Recommendations:

Complete	57	*Rosecheckers* catches all violations of these rules
Partial	45	*Rosecheckers* catches some, but not all violations of rules
False-Positives	9	These rules could be checked by *Rosecheckers*, but they will also catch some false positives
Potential	29	These rules are not checked by *Rosecheckers*
Undoable	32	These rules could not be checked by *Rosecheckers* due to various limitations in ROSE
Unenforceable	48	These rules could not be checked by any tool that relies purely on static analysis
TOTAL	220	

6.1.3 - Taxonomy

The rules and recommendations are subdivided into the following macro areas:

- Preprocessor (**PRE**)
- Declarations and Initialization (**DCL**)
- Expressions (**EXP**)
- Integers (**INT**)
- Floating Point (FLP)
- Arrays (**ARR**)
- Characters and Strings (**STR**)
- Memory Management (**MEM**)
- Input Output (FIO)
- Environment (ENV)
- Signals (SIG)
- Error Handling (**ERR**)
- Application Programming Interfaces (API)
- Concurrency (CON)
- Miscelaneous (**MSC**)

- POSIX (POS)
- Microsoft Windows (WIN)

Each rule/recommendation identifier is given by the macro area identifier followed by a numeric suffix.

Numeric values in the range of **00-29** are reserved for **recommendations**, while values in the range of **30-99** are reserved for **rules**.

The last version of the standard (v2) as given by the SEI website is composed by **114 rules** and **186 recommendations**, and is still under construction (20 rules less than how many declared by the *Rosecheckers* table above).

Because of the rules and recommendations number, and the already attained compliance to the majority of them, this document does not enumerate each single rule, but only those where at least a violation was found.

As a reference, the complete rules list is freely available from the SEI website :

https://wiki.sei.cmu.edu/confluence/display/c/

During the analysis, to be formally compatible with the MISRA C section, we'll use the *Required* keyword in place of *Rule* and the *Advisory* keyword in place of *Recommendation*.

6.1.4 - Declarations and Initialization

DCL00 (Advisory) *Declare immutable objects using `const` or `enum`.*

False positive

A false positive is signaled by this code snip:

```
uint32_t virt;
fault_addr_get(virt);
```

The tools suggests to declare the `virt` variable as `const`, thinking that its value never changes. But the `fault_addr_get` macro function contains some `inline` assembly code to extract the *page fault* address from the dedicated register and assigns it to the variable.

Project deviation

In general, is correct to declare every variable that never changes as `const`. In some occasions this is what is informally known as *syntactic sugar, t*hat is, a formally correct detail that does not change the overall functional behavior.

Follows a non compliant code snip found in the paging code.

```
#define DIR_INDEX(virt) ((uint32_t)(virt) >> 22)
unsigned int di = DIR_INDEX(virt);
```

Obviously, declaring the variable as `const` ensures that its value will never change and may generate more optimal code by some less smart compilers.

DCL01 (Advisory) - *Do not reuse variable names in subscopes.*

Some function parameters were declared using the same variable name as some global structures.

```
struct inode { … };

void function(struct inode *inode) { ... }
```

The parameters identifiers were properly renamed to don't clash with global names.

DCL02: (Advisory) - *Use visually distinct identifiers.*

The tool triggers five violations caused by the following identifiers: `current` (scheduler.c), `ktask` (scheduler.c), `need_resched` (isr.c), `timer_ticks` (timer.c) and `tss` (task.c).

From our perspective the identifiers are visually distinct, thus the violations are ignored.

DCL05 (Advisory) - *Use `typedefs` to improve code readability.*

There was a function prototype with two function pointer parameters.

```
struct slab_cache *slab_cache_create(const char *name, size_t size,
                             unsigned int align, unsigned int flags,
                             void (*ctor)(void *), void (*dtor)(void *));
```

The prototype has been modified to use types defined via the `typedef` keyword.

```
typedef void (* slab_obj_ctor_t)(void *obj);
typedef void (* slab_obj_dtor_t)(void *obj);

struct slab_cache *slab_cache_create(const char *name,
          size_t size, unsigned int align, unsigned int flags,
          slab_obj_ctor_t ctor, slab_obj_dtor_t dtor);
```

DCL13 (Advisory) - *Declare function parameters that are pointers to values not changed by the function as `const`.*

This rule is technically equivalent to the MISRA 2012 Rule 8.13.

<u>Specific deviation</u>

Functions whose pointers are meant to be assigned to a VFS (virtual file system) interface structure (e.g. `struct inode_ops`) should respect the types defined by such structures members. Thus, it is possible that a function parameter is not declared as `const` even if is not changed within the function body just to respect the interface requirements and avoid assignment casts.

```
static struct inode *devfs_lookup(struct inode *dir, const char *name)
{
   /* dir parameter not changed here */
}

/* respect the inode_ops members types to avoid casts */
static const struct inode_ops devfs_inode_ops = {
    .read    = devfs_inode_read,
    .write   = devfs_inode_write,
    .mknod   = devfs_mknod,
    .lookup  = devfs_lookup,
};
```

6.1.5 - Expressions

EXP05 (Advisory) - *Do not cast away a `const` qualification.*

This rule is technically equivalent to the MISRA 2012 Rule 11.8.

False positives

The tool signals some false positives all caused by code snips similar to the following:

```
int pipe_write(struct inode *inod, const void *buf,
               size_t count, off_t off)
{
    const char *ptr = (const char *)buf; /*** Here ***/
   …
}
```

In particular, after some tests, looks like the tool doesn't like when a "`const void`" pointer is casted into a "`const char`" pointer.

EXP11 (Advisory) - *Do not apply operators expecting one type to data of an incompatible type.*

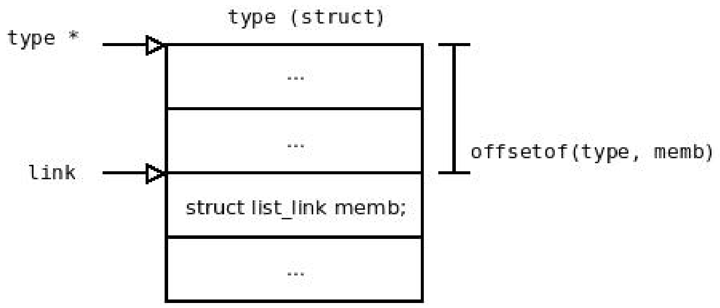
Specific deviation

Almost all the violations were caused by usage of the following widespread macro:

```
#define list_container(link, type, memb) \
    ((type *) ((char *)(link) - offsetof(type, memb)))
```

After some tests we've realized that the error was due to the conversion from a "`char`" pointer to a "`type`" pointer.

The warning is indeed legitimate because is a risky operation and the cast can generally cause alignment issues. But in our case is absolutely safe to do so; indeed the macro returns a pointer to an already instanced variable of type "`type`" and located at the address returned by the macro.



Project deviation

Is forbidden to convert an integer constant to a pointer type.

```
#define PAGE_TAB_MAP    0xFFC00000
tab = (uint32_t )PAGE_TAB_MAP;
```

Unfortunately, within the lowest level kernel code, this type of conversion is sometimes unavoidable (e.g. memory mapped device registers and paging).

EXP32 (Required) - *Do not cast away a `volatile` qualification.*

There were two instances of a `volatile int` variable casted to an `int`. Fixed.

EXP36 (Required) - *Do not cast between pointers to objects or types with differing alignments.*

Generally the warning was caused by `kmalloc` result pointers that were not casted to the destination pointer type.

False positive

In “arch/task.c”, the following code snip is signaled as a violation

```
struct task_ifr { uint32_t a, b, c; } ifr;
uint32_t *p = (uint32_t *)&ifr;
```

The cast can be assumed to be safe since the structure member types are equal to the destination pointer type. The above code snip is technically equivalent to

```
uint32_t ifr[3];
uint32_t *p = ifr;
```

False positive

In “vfs.c”. Given this prototype

```
struct htable_link *htable_lookup(struct htable_link * const *htable,
                                  long long key, unsigned int bits);
```

The tool signals an exception in this assignment

```
struct htable_link *lnk;
lnk = htable_lookup(inode_htable, KEY(dev,ino), INODE_HTABLE_BITS);
```

a false positive is assumed.

6.1.6 - Integers

Compliance with this section is the most hard to obtain because of the violations, when present, are not isolated but widespread project-wide because of code design or stylistic motivations.

Advisory recommendations that requires too much invasive interventions or code refactory are left *"as-is"*.

INT01 (Advisory) - *Use `rsize_t` or `size_t` for all integer values representing the size of an object.*

Project deviation

Represent a size of an object directly with an "`unsigned int`" is a very widespread and, generally, accepted practice.

Huge intervention is required for compliance. Code is left *"as-is"*.

INT07 (Advisory) - *Use only explicitly `signed` or `unsigned char` type for numeric values.*

Project deviation

Buffers that are supposed to contain ASCII encoded names were declared as simple char arrays. The sign is not meaningful in this context.

INT13-C: (Advisory) - *Use bitwise operators only on unsigned operands.*

False positives

The tool generates a lot of false positives here, thus we've decided to resolve just few cases and leave alone the rest (also considering that this is an advisory rule).

For example, just the following "random.c" function generates 11 warinings:

```
static uint32_t rand_get(void)
{
    uint32_t r;

    r = ((z[0] << 6) ^ z[0]) >> 13;
    z[0] = ((z[0] & 4294967294UL) << 18) ^ r;
    r = ((z[1] << 2) ^ z[1]) >> 27;
    z[1] = ((z[1] & 4294967288UL) << 2) ^ r;
    r = ((z[2] << 13) ^ z[2]) >> 21;
    z[2] = ((z[2] & 4294967280UL) << 7) ^ r;
    r = ((z[3] << 3) ^ z[3]) >> 12;
    z[3] = ((z[3] & 4294967168UL) << 13) ^ r;
    r = z[0] ^ z[1] ^ z[2] ^ z[3];
```

```
        return r;
    }
```

After some investigations we've discovered that the source of the problems is that we've used the `uint32_t type` in place of `unsigned int`. The warning disappears as soon as we replace one type with the other.

INT14 (Advisory) - *Avoid performing bitwise and arithmetic operations on the same data.*

In low level code is very common to mix bitwise operations with arithmetic.

Project deviation

Most of the violations were caused by the `ALIGN_UP` and `ALIGN_DOWN` macros. These two function-like macros, allows to align a value to a given (next or previous) power of two value boundary and are very important in kernel code that handles memory addresses (e.g. stack pointers).

```
#define ALIGN_UP(val, siz)  (((val) + ((siz) - 1)) & ~((siz) - 1))
#define ALIGN_DOWN(val, siz) ((val) & ~((siz) - 1))
```

We've not found another efficient way to do the same job without mixing arithmetic and bitwise operators.

Specific deviation

Consider the initialization of a page table. We need to set each entry to the mapped physical address ORed with some access flags.

```
for (i = 0; i < 1024; i++)      /* Identity map the first 4 MB */
    tab[i] = (i * PAGE_SIZE) | PTE_W | PTE_P;
```

Obviously, to suppress the warning, given that PAGE_SIZE is a power of 2, the multiplication part can be replaced with a left shift `(i << 12)` but we think that the current code version is more clear about what it functionally does.

Some other warning instances of this type were present and, given the advisory nature of the rule, we've decided to leave almost all the code untouched.

INT33 (Required) - *Ensure that division and modulo operations do not result in divide-by-zero errors.*

There was a division by an input parameter coming from a not trusted source within the `zone_init` memory management function.

False positive

Even after the fix, a false positive is still signaled.

```
if (frame_size == 0) /* Added check */
    return -1;
...
return buddy_init(&ctx->buddy, size / frame_size, frame_size); /** Here **/
```

Looks like the warning disappears if we move the check just before the return statement.

INT34 (Required) - *Do not shift a negative number of bits or more bits than exist in the operand.*

False positives

Four warnings were triggered by the following code snip:

```
#define SIGSYS    31
#define SIGUNUSED SIGSYS

#define NSIG      (SIGUNUSED + 1)

typedef unsigned long sigset_t;

#define sigaddset(set, n) \
          ((0 < (n) && (n) < NSIG) ? ((*(set) |= (1 << (n))), 0) : -1)
```

Even if we usually skipped discussions about warnings coming from macros defined within the standard C library, like `sigaddset`, we are going to discuss it here because of a false positive.

The tool complains that, before assignment, we should check that `(1<<n)` fits in a sigset_t variable. Indeed we don't do any comparison against the `sizeof(sigset_t)` but we check that `n` is less that `NSIG` (i.e. 32) and is guaranteed (by the C standard) that an `unsigned long` is at least 32 bits so no further checks were performed.

False positives

There are four violations because of the risk to perform a left/right shift for more bits than how many are available on the operand.

Actually the condition cannot happen because the shift quantity was already checked. The tool is not able to notice it.

```
size_t ind, shift;
shift = 10 + sb->log_block_size;
if (shift >= 8 * sizeof(size_t))
    return -1;  /*** Enforces the shift safety ***/
...
ind = offset >> shift;   /*** Warning given Here ***/
```

INT36 (Required) - *Take care when converting from pointer to integer or integer to pointer.*

The architecture independent code is compliant.

Within the architecture dependent code and device drivers is easier to find code that contains casts from (fixed width) integers to pointers and vice-versa. This is generally safe, enforced by a context specific knowledge of the pointer size. E.g. in a 32-bit machine is safe to cast a pointer to and from a `uint32_t`.

As a reference, some specific deviations are given, the others are similar. All the deviations similar to the given examples are just ignored.

Specific deviation

Within paging code physical addresses are handled as uint32_t quantities while virtual addresses are handled as pointers. This is a practical choice to simplify some implementation operations.

For example, a page table is defined as an array of physical addresses OR-ed with some flags. It is a lot easier to manage it as an array of uint32_t in place of an array of pointers to uint32_t (also considering that OR-ing pointers and integers is forbidden by another rule...).

Specific deviation

Very often, device drivers uses memory mapped registers and data. In our case the VGA video memory buffer is memory mapped at a fixed address, defined as `VIDEO_BUF` with the address literal value. In this occasion, a casts from integer to a pointer cannot be avoided.

```
#define VIDEO_BUF  (KVBASE + 0xB8000)
uint16_t *buf = (uint16_t *)VIDEO_BUF;
```

Specific Deviation

The `execve` syscall generic implementation contain some architecture specific code. The implementation assumes 32-bit addresses in more than one location mainly caused by the 32-bit specific elf program headers.

A non trivial code re-engineering is required here and was already within the kernel TODO list.

Interesting fix

A more subtle and particular violation was present. Because of its nice characteristics an in-depth violation description along with the fix is given.

The `getcwd` syscall is defined to return a `char` pointer on success and a `NULL` pointer on failure. The syscall implementation, as all the others, in kernel space should be able to pass, somehow, the `errno` value to the user space. The adopted general strategy is to return a negative `errno` value as a result when the syscall fails.

In user space syscall wrapper, if the kernel syscall returns a negative value, the `errno` global variable is set to the negative of such a value before returning to the caller.

In the `getcwd` implementation the thing is a bit tricky. Indeed the negative value is casted to a char pointer and the standard C library should be able to see if that value is indeed an error or is a valid pointer value.

The required condition is that the negative of every possible value is different from every possible buffer pointer value. Assuming `sizeof(uintptr_t)=4` and a max `errno` value of 131 (as with our current implementation) the condition translates to

$$bufptr \notin [-1, -137] = [0xFFFFFFFF, 0xFFFFFF7D]$$

With our current x86 implementation the user space lives below the kernel start virtual address *0xC0000000* thus the condition always holds. Anyway, speaking in general, this is not a very safe and portable solution.

The syscall implementation code has been modified to return an `int` in place of a pointer to `char`. The POSIX compliance of `getcwd` is then maintained by adapting the user space wrapper.

```
/* Kernel space implementation */
int sys_getcwd(char *buf, size_t size)
{
    int res;
    ...
    return dentry_path(current->cwd, buf, size);
}

/* User space wrapper */
char *getcwd(char *buf, size_t size)
{
    char *ret = buf;
    if (syscall(__NR_getcwd, buf, size) < 0)
        ret = NULL;
    return ret;
}
```

6.1.7 - Arrays

ARR02 (Advisory) - *Explicitly specify array dimensions, even if implicitly defined by an initializer.*

The arrays that are used only within a translation unit were not declared with an explicit width.

The declaration of the array was indeed following the widespread form:

```
struct mystruct myarray[] = { … };
#define MYARRAY_LEN (sizeof(myarray) / sizeof(myarray[0]))
```

The code has been changed to be compliant with this rule.

ARR30 (Required) - *Guarantee that array indices are within the valid range.*

There were four violations all triggered by the same motivation. A file descriptor, i.e. a signed integer, coming from user space was not tested whether it was greater than 0 before being used as an array subscript.

False positive

Even after the corrections, one violation is signaled within the `dup2` syscall implementation.

```
int sys_dup2(int oldfd, int newfd)
{
    int status;

    if (oldfd < 0 || oldfd >= OPEN_MAX || newfd < 0 || newfd >= OPEN_MAX ||
            current_task->fds[oldfd].fil == NULL) {
        return -EBADF; /* Invalid file descriptor */
    }

    if (current_task->fds[newfd].fil != NULL) {     /*** <--- Here ***/
        ...
    }
    ...
}
```

6.1.8 - Error Handling

ERR33 (Required) **-** *Ensure that return values are compared against the proper type.*

One violation was caused by a `strlen()` return value compared against a `uint8_t`.

False positives

Four false positives are left. All the signaled violations are caused by code like the following:

```
size_t seed_siz;
...
if (seed_siz == 0)
    return -1;
```

This is not a function return value. This is an unsigned integer compared against a signed literal value, a violation already discussed within the MISRA "Essential type model" chapter.

6.1.9 - Miscelaneous

MSC01 (Advisory) – *Strive for logical completeness.*

One violation was present and caused by a switch without a default case. Trivially fixed in the MISRA analysis by adding an empty default case.

MSC05 (Advisory) **-** *Do not manipulate time_t typed values directly.*

There were seven violations to this recommendation and all were in the `nanosleep` syscall implementation.

The solution to this issue is a cast from `time_t` values to `unsigned long` before perform any arithmetic or logical operation.

MSC12 (Advisory) - *Detect and remove code that has no effect.*

The macros va_start and va_end were containing a statement with no effect, e.g. the va_start has bee redefined from:

```
#define va_start(dst, src) ((void) ((dst) = (va_list)(src)))
```

To:

```
#define va_start(dst, src) ((dst) = (va_list)(src))
```

Another source of violation was the `sigemptyset` macro, redefined from:

```
#define sigemptyset(set) (*(set) = 0, 0)
```

To:

```
#define sigemptyset(set) (*(set) = 0)
```

6.2 Conformance Matrix

Static Analysis Tools: *Rosecheckers with ROSE v0.9.5a*

First pass violations: violations found in the POST MISRA-C 2012/2004 analysis.

Final pass violations: violations found in the final codebase (v0.2.0).

Final pass ignored: false positives.

	Type	**First Pass Violations**	**POST-MISRA Violations**	**Final Pass Violations**	**Final Pass Ignored**	**Check Method**
DCL00	Advisory	35	25	11	1	Rosecheckers
DCL01	Advisory	62	19	0	0	Rosecheckers
DCL02	Advisory	3	5	0	5	Rosecheckers
DCL05	Advisory	2	2	0	0	Rosecheckers
DCL13	Advisory	6	5	0	5	Rosecheckers
EXP05	Advisory	3	5	0	5	Rosecheckers
EXP11	Advisory	67	72	70	0	Rosecheckers
EXP12	Advisory	44	1	0	0	Rosecheckers
EXP36	Required	24	26	0	2	Rosecheckers
INT01	Advisory	79	84	92	0	Rosecheckers
INT07	Advisory	17	17	16	0	Rosecheckers
INT13	Advisory	58	104	15	16	Rosecheckers
INT14	Advisory	24	27	29	0	Rosecheckers
INT33	Required	2	1	0	1	Rosecheckers
INT34	Required	8	9	0	8	Rosecheckers
INT36	Required	26	25	3	19	Rosecheckers
ARR02	Advisory	3	3	0	0	Rosecheckers
ARR30	Required	7	4	0	1	Rosecheckers
ERR33	Required	3	5	0	5	Rosecheckers
MSC01	Advisory	1	0	0	0	Rosecheckers
MSC05	Advisory	3	7	0	0	Rosecheckers
MSC12	Advisory	6	10	0	0	Rosecheckers

6.2.1 - Recap

Section	Subject	First Pass Violations	Post MISRA Violations	Final Pass Violations	Final Pass Ignored
DCL	Declarations and Initialization	108A	56A	11A	11A
EXP	Expressions	114A + 24R	73A + 26R	70A	5A + 2R
INT	Integers	178A + 36R	232A + 35R	152A + 3R	16A + 28R
ARR	Arrays	7A + 6R	5A + 4R	0	1R
ERR	Error Handling	3R	5R	0	5R
MSC	Miscelaneous	10A	17A	0	0

Note that in the table the required and advisory violations are separated. For example: 7A + 6R means that there are 7 violations to advisory rules and 6 deviations to required rules.

7 Final Considerations

Coding standards mitigate the effects that the implementation-defined, unspecified and undefined behaviors of a programming language have on its performance and quality.

While they are of benefit to all projects, some of the more onerous guidelines can be overkill for projects with lower integrity requirements. A coding standard subset can be used to reduce the severity of the standard in a reasonable and controlled way, allowing it to be applied more cost effectively to a wider range of projects.

In either case, <u>the decision must be made early</u> in the development process to save development time and costs involved in changing direction after coding has begun.

At the same time, the standard or subset must be analyzed for its ability to be checked for compliance, and appropriate tools should be identified and deployed up front, rather than waiting until the code has already been written and tested, when standards exceptions have already proliferated.

Developers can avoid generating extensive documentation to provide evidence that noncompliance doesn’t affect system behavior, and will be less likely to have to substantially rework code, with the risk of introducing defects after the code has been tested.

> In our kernel analysis the risk to introduce errors within already tested code was very often higher than the benefit gained by some rule.

By coding to a standard or subset from <u>the beginning</u> of a project, and using tools that automatically verify and enforce those rules, developers can save development time and effort.

Re-engineering of a relatively *big* project could be a very daunting task and in some cases the probability to introduce new errors is too high to be worth the risk[3].

In the end, even though we cannot claim complete compliance with the MISRA or CERT C coding standards, the conducted analysis given its fruits and showed to be useful for the following reasons:

- The gap to compliance is now smaller and we’ve provided the motivations behind each violation class.
- Several – dormant – bugs were found and fixed (dormant faults).
- A precious opportunity for code audit and refactory. These opportunities (especially in professional contexts) are very rare unless there is a valid motivation behind them.
- An opportunity to learn something new about the C language subtle details and traps.

3 A solid unit tests collection could greatly reduce this risk