LDRA Software Technology

Technical White Paper

Implementing CERT C with the LDRA tool suite®

Eliminating insecure coding practices and undefined behaviours that can lead to exploitable vulnerabilities and unreliable applications

www.ldra.com

* Registration required to download the document

Contents

Introduction

The CERT C Coding Standard consists of a set of guidelines designed to assist in the development of safe, reliable, and secure systems. It was developed by the Computer Emergency Response Team (CERT) division of the Software Engineering Institute (SEI), with reviews and contributions from many industry experts[1]. SEI is a federally funded pioneer of programming practices, based at Carnegie-Mellon University with support from the US Department of Defense.

SEI's CERT division was formed in response to the Morris worm incident of 1988, and focuses on improving the security and resilience of computer systems and networks. CERT is well respected for its expertise and works closely with organisations from both private and public sectors including Cisco, Oracle, and the US Department of Homeland Security. The idea of a secure coding standard for the C language was born in 2003 of CERT's discussions with the ISO/IEC JTC1/SC22/WG 14 committee (WG14) responsible for the C language standard.

Most recently, WG14 has organised a technical committee of industry experts including representatives of static analyser vendors such, as LDRA, to formulate the ISO/IEC TS 17961:2013 standard which is designed to establish a baseline set of requirements for static analysis tools and C language compilers. All ISO/IEC TS 17961:2013 rules are designed to be enforceable by static analysis, while encompassing many of the CERT C guidelines.

Static Analysis

There are software test tools available to perform a variety of tasks, which work in a variety of different ways. Examples include Unit Test tools, Code Coverage tools, Penetration Test tools and tools to check for compliance with coding rules and recommendations, such as those specified in CERT C.

One of the key differentiators between tools is that some perform dynamic analysis, while others perform static analysis. Static analysis is performed without actually executing the software, whereas dynamic analysis involves the execution of all or some of that software. Tools used to check coding rules are concerned with the source code rather than its execution, and so use static analysis techniques.

CERT C

Although it has been twice published in book form, the CERT C coding standard continuously evolves to incorporate the comments and suggestions of C language experts and so the primary reference is CERT's secure programming wiki. References to the standard in this white paper were accurate at the time of writing.

The primary aim of this evolution is to ensure that the CERT C coding standard promotes current best practices at all times with due regard to safety, reliability and security. These best practices are determined through a risk assessment which measures:

- The likelihood of a violation of the rule causing a defect
- The severity of the defect
- The cost of fixing the defect

In addition to hosting the standard itself, the wiki provides a forum for debate amongst contributors. This provides a vehicle for continuous improvement, minimising controversial rules by ensuring that the guidelines are precisely phrased and do not unfairly exclude good code. The resulting CERT C standard is therefore based on consensus in the general programming community.

Some standards include coding style guidelines to aid readability and maintainability, providing rules for such as tab indentation and bracket alignment. These considerations are specifically outside the scope of CERT C which only impacts on coding style where it can lead to defects. For instance, the lowercase letter 'l' is easy to mistake for the number 1 in many fonts, so the CERT C rule DCL16-C. "Use 'L,' not 'l,' to indicate a long value" is designed to deal with that.

[1] https://www.securecoding.cert.org/confluence/display/c/Acknowledgments

The terminology for coding guidelines varies from one standard to another. The CERT C coding standard subdivides its guidelines into “rules” and “recommendations”. For a guideline to be defined as a rule, it is required to meet three conditions.

- Violation of the guideline is likely to result in a defect that may adversely affect the safety, reliability, or security of a system, for example, by introducing a security flaw that may result in an exploitable vulnerability.
- The guideline does not rely on source code annotations or assumptions.
- Conformance to the guideline can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Guidelines are defined as recommendations when they fail to meet one or more of these conditions but are still likely to improve the safety, reliability, or security of software systems. For example, in the miscellaneous section of the CERT C standard there are recommendations on the use of both code comments (MSC04-C “Use comments consistently and in a readable fashion”) and compiler warnings (MSC00-C “Compile cleanly at high warning levels”).

Static Analysis and CERT C

LDRA tools statistically analyse code to detect a range of conditions, a subset of which are applied to confirm compliance with CERT C guidelines:

- Checking for dangerous expressions
- Anomaly detection
- Type conversion and cast-usage analysis
- File handling analysis
- Control and data flow analysis
- Function call analysis
- Parameter analysis
- Array declaration analysis
- Identifier usage analysis
- If statement analysis
- I/O routine analysis
- Jump analysis
- Library function usage analysis
- Literal value analysis
- Loop analysis
- Preprocessor usage analysis
- Pointer analysis
- Statement analysis
- Structure usage analysis
- Coding style analysis
- Switch statement analysis
- Variable type analysis

In practice this means that rules are required to clearly define the restrictions on code use, and that the restrictions must be directly associated with a risk. In contrast, recommendations often encompass general best practices, or identify situations which may sometimes be dangerous.

There are exceptions to both rules and recommendations when good programming practice permits. For instance, CERT C rule ERR33-C “Detect and handle standard library errors” recognises that there may be some cases where checking the error code in a function return may be unnecessary, but dictates that in such cases the return value is explicitly discarded by casting the value to void.

In addition to the distinction between rules and recommendations, CERT C is divided into 17 categories. These range in scope from guidelines relating to the preprocessor, through language characteristics such as integers and floating points, to POSIX and Windows considerations. Clearly, the CERT C guidelines do not focus only on the language syntax and are intended to be used selectively - CERT C rules concerning POSIX or Windows operating systems have no importance when dealing with a bare-metal application, for example.

A similar issue results from CERT C being forward-looking and therefore advocating the use of features which may not be available on all compilers. For example, the “static_assert” facility in the C language is supported by relatively few compilers at the time of writing and yet the CERT C standard includes the guideline DCL03-C “Use a static assertion to test the value of a constant expression”. This clearly cannot be applied if the compiler in use does not support it. In figuring out which rules make sense for a particular program, a valuable resource is the latest available edition of the associated text book[2] or (preferably) the CERT C wiki[3]

[2] Secure Coding in C and C++ (SEI Series in Software Engineering) 2nd (second) Edition by Seacord, Robert C. published by Addison Wesley (2013)
[3] https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard

Sources of Guidelines for Detecting Vulnerabilities

Although the C standards[4] are extensive in defining how compiled C language should behave, they are by no means exhaustive. One of the challenges inherent in developing secure C code surrounds the issue of the remaining "undefined" or "unspecified" behaviour. Compliers are permitted flexibility in how such behaviour is to be handled, and the resulting unpredictability undermines CERT C's goal of safe, reliable and secure systems.

There is a group of guidelines in the CERT C standard to deal with this behaviour, with examples appearing in the sections dealing with integers, characters, strings, arrays, and declarations. In this respect, CERT C shares much in common with other coding standards such as MISRA C, which can be cross-referenced to many CERT C rules. That said, CERT C only restricts the use of "undefined" or "unspecified" behaviours which have risks associated with them.

Another group of guidelines was derived from the work of the MITRE Corporation (MITRE) who focus on a number of related fields including cybersecurity[5]. Their related research gave rise to both the CVE database of known vulnerabilities in software, and the CWE database of known software weaknesses allowing such vulnerabilities to occur.

These weaknesses and vulnerabilities are collated from a variety of sources across the software security industry. CERT draws on MITRE's expertise by citing and cross referencing MITRE CVE and CWE entries in the CERT wiki. Reciprocally, CERT guidelines are also cited in the CWE database in an effort to combat weaknesses.

Verifying Compliance with the CERT C Secure Coding Standard

The ever-increasing complexity and connectivity of embedded applications represents a worsening headache for developers aiming to ensure that their products are safe and secure. Although they can develop their software in accordance with the CERT C standard, manual review to check for application of the guidelines is really only practical for small code bases. An automated tool to detect violations with reference to the coding standard makes things more straightforward.

LDRA's static analysis tools hold a superset of the rules and guidelines associated with CERT C and other standards, and combine a detailed analysis of the C-syntax with the analysis of abstract data and control flow models. Such an approach permits two levels of abstraction, ensuring that checks are made with reference to not only the dangerous details of the C language, but also in the context of the program as a whole. Figure 1 shows how a specific coding standard such as CERT C is selected.

These tools implement a number of analysis phases, namely

- Main Static Analysis, including variable declaration, macro usage, function call usage, control flow, and local data flow
- Data Flow Analysis Phase, for the analysis of system-wide data flow
- Cross Reference Analysis Phase, to build on the data flow analysis by cross-referencing the symbols used throughout the system

Those phases combine to apply a catalogue of different checks, a subset of which is applied to confirm compliance with CERT C guidelines (sidebar).

[4] https://en.wikipedia.org/wiki/ANSI_C
[5] https://www.mitre.org/capabilities/cybersecurity/overview?category=all

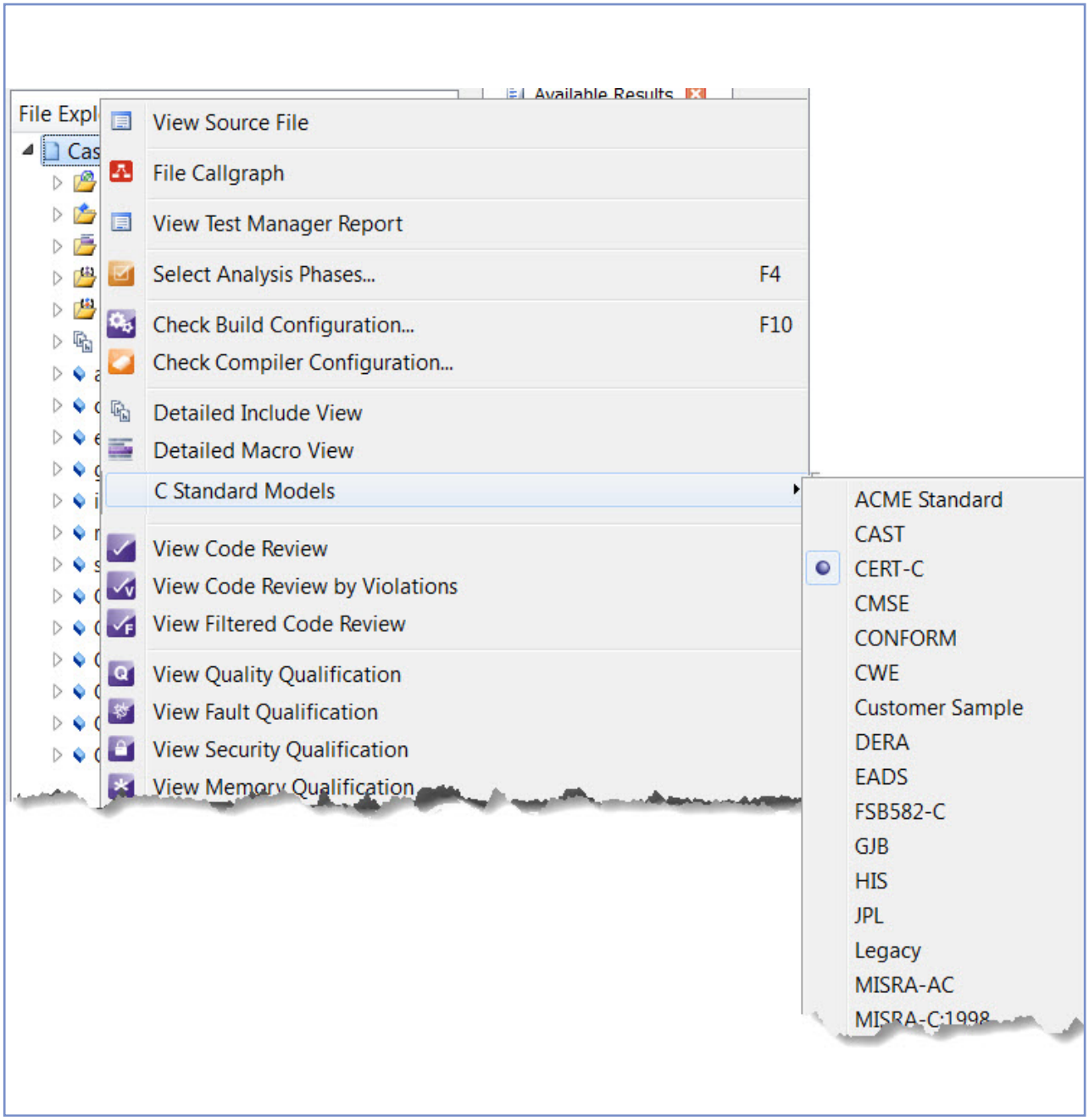


Figure 1: Selecting the appropriate CERT C standard in LDRA TBvision

Main Static Analysis Phase

During the Main Static Analysis Phase, the code is parsed to enable the tool to check for problems with variable declaration, macro usage, function call usage, control flow, and local data flow.

Expression-Level Data

CERT C Rules concerned with the use of expression-level data are verified during this phase. Examples include *ARR37-C "Do not add or subtract an integer to a pointer to a non-array object"* (Figure 2) and *EXP45-C "Do not perform assignments in selection statements"*.

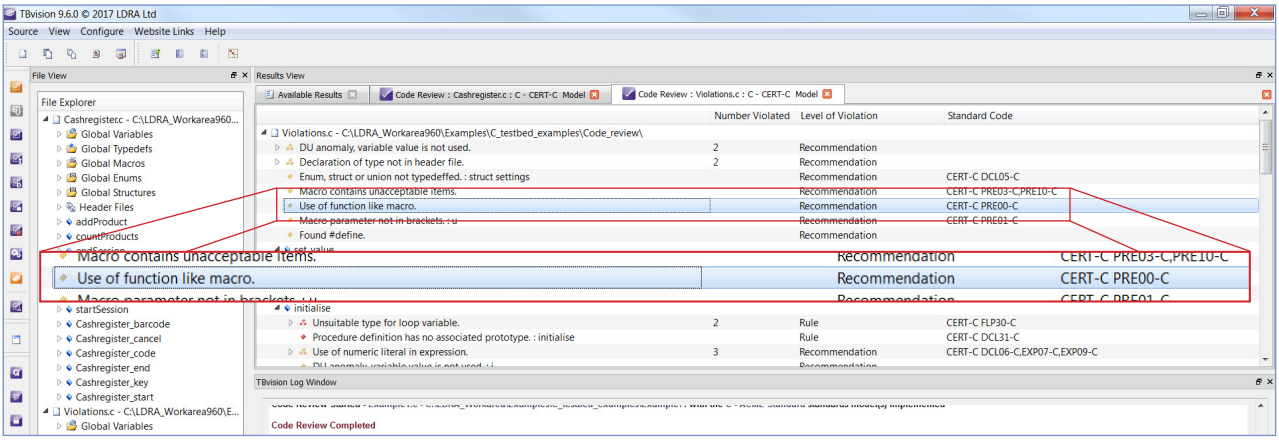


Figure 2: Reporting a violation of CERT C PRE00-C – "PRE00-C. Prefer inline or static functions to function-like macros"

Control Flow

In addition to reporting on these syntactical issues, the Main Static Analysis phase also establishes an understanding of the control flow paths within the code. This provides the information for the tool not only to present those flow paths in graphical form, but also to generate a reformatted version of the code such that there is only one instruction per line.

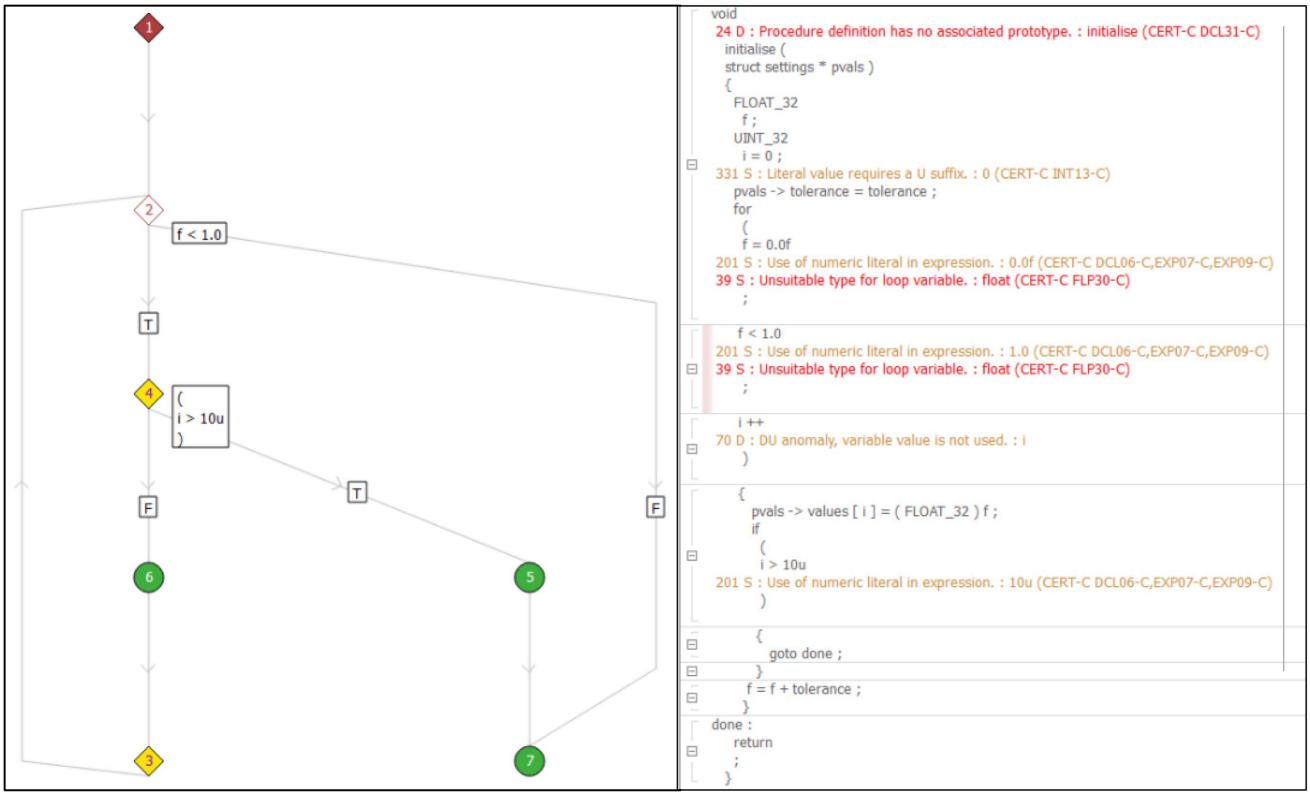


Figure 3: Presenting the control flow in graphical form, alongside reformatted source code

It also provides the underlying information for additional CERT C guideline checks. One such example is the recommendation MEM05-C "Avoid large stack allocations." MEM05-C includes the suggestion that recursion should be avoided because neither the control flow nor the resource use associated with such a mechanism is easily understood (Figure 4).

```
unsigned long fib1(unsigned int n)      /*Non compliant 6 D*/
{
  if (n == 0) {
    return 0;
  }
  else if (n == 1 || n == 2) {
    return 1;
  }
  else {
    return fib1(n-1) + fib1(n-2);
  }
}

void main(){

}
```

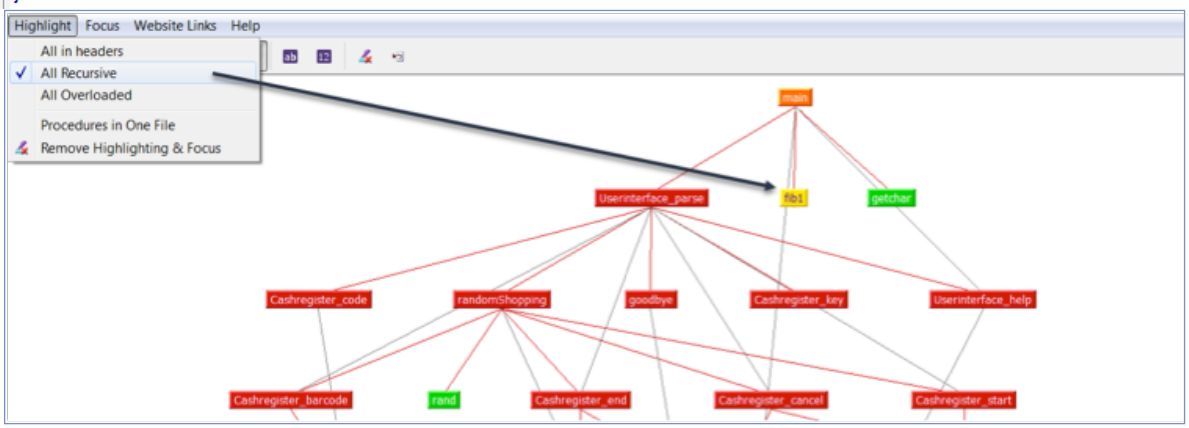


Figure 4: The system call graph showing the use of recursion, an example of a violation of CERT C "MEM05-C – Avoid large stack allocations"

Data Flow Analysis Phase

This second static analysis phase collates information relating to the system wide data flow, complementing the reformatted source code and the control flow to complete a 3 dimensional model. The combination of these elements provides the basis of a mechanism to check variable use in complex control flow scenarios, and in the context of CERT C, to check for the violation of rules such as *EXP30-C "Do not depend on order of evaluation between sequence points"*.

It is interesting to reflect on how that is approached automatically, and hence to consider the effort that would be required to verify compliance through manual code review.

A sequence point is a point in program execution at which all side effects are evaluated before going on to the next step. They are often mentioned in reference to C and C++, because they are a core concept for determining the validity and, if valid, the possible results of expressions.

As a practical example, consider two functions *f()* and *g()* in C or C++.

Scenario 1: Expression *f()+g()*

The plus operator is not associated with a sequence point, and therefore in the expression it is possible that either f() or g() will be executed first.

Scenario 2: Expression *f(),g()*

The comma operator is associated with a sequence point, and therefore in the code f(),g() the order of evaluation is defined: first f() is called, and then g() is called[6].

[6] http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html

The point of the rule is to ensure that the outcome of an expression such as that in Scenario 1 does not depend on what happens before it is executed. In order to be certain of that, it is necessary to consider the different potential orders of evaluation between sequence points. This includes situations such as the use of unary operators outside of a standalone expression, the use of volatile variables in complex expressions, the use of assignment in an expression that references the variable assigned within the same sequence point, and parameter side effects of function calls.

To do that manually would be time consuming, demand a very high level of expertise and concentration, and even then would be subject to human error. Automating the process is a far more pragmatic approach, because an analysis tool is able to recognize these situations by analysing not only what operators and variables are being used, but also the context of the expression structure and how variables in that expression might be affected by prior use.

This example involves just one error, but languages like C and C++ can include multiple constructs on the same line of code, and indeed, single constructs can encompass multiple violations. Recognising these concurrent instances of violations is by no means a straightforward task for code review, and reinforces the case for automation.

That benefit becomes even more transparent in the wider context of library functions and system calls. An automated analysis can consider the handling of return values and parameters associated with system calls, and the possible misuse of library functions. For instance, in order to check for violations of CERT C rule ARR38-C "Guarantee that library functions do not form invalid pointers", it is necessary to check that system calls writing to memory are provided with arrays of adequate size.

It can also check how file pointers are used by library functions; just one important aspect of the CERT C recommendation MSC15-C "Do not depend on undefined behaviour". There are several undefined behaviour cases associated with file pointers, including (for example) what should happen if file pointers are closed more than once.

Cross Reference Analysis Phase

Once the data flow picture is complete, it is possible to build on that information by cross- referencing the symbols used throughout the system and hence answer questions such as

- Where is a variable used? In which functions, and which files?
- Where is that variable assigned values?
- Where is its value used in calculations?
- Is it unused, or used only once?

This information also provides a foundation for the assessment of potential mismatches between how an identifier is used in different places within a code base. For example, differences between how a symbol is defined and declared can lead to undefined behaviour, and is another example of a check relevant to CERT C recommendation *MSC15-C "Do not rely upon undefined behaviour"*.

This cross referencing also permits the extension of more perfunctory checks completed in earlier phases, such as extending the analysis of potential array usage violations to instances spanning more than one procedure, pertinent to the CERT C Rule *ARR30-C "Do not form or use out-of-bounds pointers or array subscripts"*.

Comparing Static Analysis Tools

There is a plethora of different static analysis tools on the market, and it is easy to draw up a list of requirements. Fast analysis, maximum coverage of rules and minimal false positives (incorrectly marked "failed") and negatives (incorrect "passes") are important, but sometimes differentiating factors are more subtle.

For example, consider the previous example, *ARR30-C "Do not form or use out-of-bounds pointers or array subscripts"*. It would be easy to claim compliance for that rule without the cross reference phase and its associated deeper dive into the use of arrays, even though that would lead to less complete detection. But it is actually the more subtle violations spanning several functions which are likely to be the most difficult to detect – and, unfortunately, often the most likely to be present.

Such considerations are key in differentiating between "lightweight" tools, and "heavyweight" ones such as the LDRA tool suite. Both have their place and a fast, inexpensive tool is likely to be far better than using nothing at all, and sometimes speed is important. For example, a common strategy is to use a lightweight tool (or a heavyweight tool in lightweight mode) for quick developer feedback, and then to the heavyweight tool for more detailed code review.

It is also undeniably true that neither is infallible. However, the thoroughness of the analysis of "heavyweight" tools means that where they do provide inaccurate information, it will almost always be in the form of warnings that err on the safe side – that is, false positives for assessment. That is a much safer position to take than the false negatives more commonplace in the lightweight tools, where there are violations but no warning of them.

In this context, it is useful to consider a summary of the information associated with Automated Detection tools available on the CERT website[7].

The analysis shows a total of 306 CERT C guidelines. Putting aside the reservations relating to incomplete detection illustrated by the *ARR30-C* example above, the data in Figure 4 and Figure 5 was collated on the basis that where there is an indication that a particular tool is able to detect a violation to any extent whatsoever, the related guideline can be considered to be implemented.

Some tool vendors have stated which of their own checks contribute to the detection of a particular CERT C violation. Others have gone further and have clarified whether the CERT C guideline coverage is full, partial or enhanced. In this case, enhanced detection means that a stricter detection is performed, producing false positives as compared with the CERT C guideline.

Implementation Tables

Tool	Known	Unknown	Total
Co. A	7	0	7
Co. B	96	0	96
Co. C	79	43	122
Co. D	50	0	50
Co. E	11	0	11
Co. F	63	0	63
Co. G	0	3	3
Co. H	36	12	48
Co. I	20	1	21
Co. J	135	0	135
LDRA tool suite	200	1	201
Co. L	10	0	10
Co. M	2	24	26

Figure 5: Summary showing the number of guidelines addressed by each tool, out of a total of 306 in the standard.

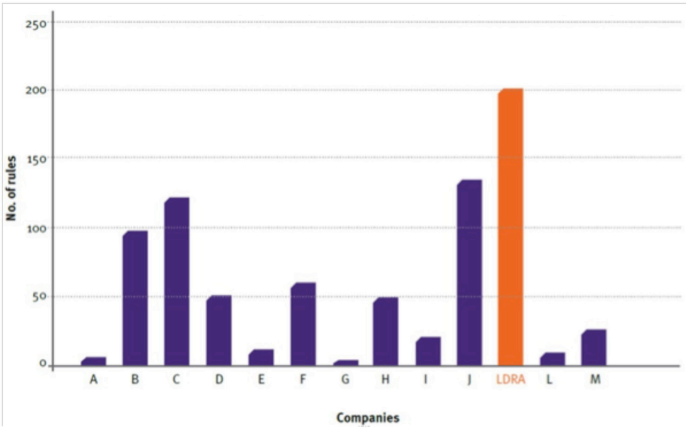
[7] www.securecoding.cert.org as of 14/09/2015

In Figure 5, the “Known” and “Unknown” categories relate to the CERT website descriptions of both the CERT C guidelines and the static analysis tools.

- The “Known” column shows the number of guidelines for which there is either a confirmation of coverage, a description of coverage, or both.
- The “Unknown” column shows the number of guidelines for which there is neither confirmation of coverage, nor a description of it.

It is estimated that at least 42 of the CERT C guidelines are not checkable by static analysis - for example, because they are not precisely defined, or because knowledge of user intent may be required for implementation. One example of the latter case is the CERT C recommendation *ARR00-C “Understand how arrays work*”. No tool can guarantee that a programmer fully understands every aspect of array functionality applicable to his program, but a static analysis tool can identify particular coding practices that suggest otherwise

Figure 6 shows a direct comparison between the analysis products listed on the CERT website.



Tool	No. of guidelines
A	7
B	96
C	122
D	50
E	11
F	63
G	3
H	48
I	21
J	135
LDRA tool suite	201
L	10
M	26

Figure 6: Tool vendor comparison for CERT C compliance

No information on the completeness of detection is available on the CERT website. However, when a tool is used in projects compliant with standards such as DO-178B/C, IEC 61508 or ISO 26262, evidence is required that the tool used can perform adequately. It may be a useful comparator to consider how extensively different tools have been used on such projects to the satisfaction of third party or client assessors

Presentation of Results

No matter how effective a particular tool is at the detection of guideline violations, if the results are not presented in an effective manner then mistakes are likely. It is therefore useful to ensure that code quality, fault detection, and avoidance measures are presented in logical, easily read formats such as graphs, flow graphs, and code review reports.

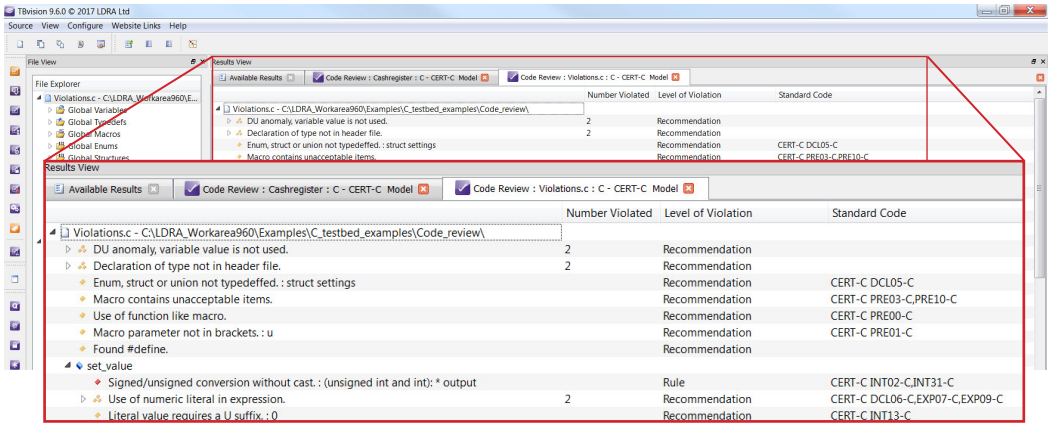


Figure 7: The TBsecure module of the LDRA tool suite checks for compliance to over 200 CERT C guidelines

A static analysis tool is likely to be only one part of a complete development toolchain, complemented by dynamic analysis and requirements traceability tools. An integrated, cohesive and intuitive environment for these components is likely to pay dividends in efficiency and in the minimisation of human error.

Conclusion

It is possible to implement the Carnegie Mellon Software Engineering Institute (SEI) CERT C Secure Coding Standard manually, and to verify that implementation by means of code review. However, such a task is onerous and prone to error, and for most of the CERT C guidelines there are established static analysis tools available which promise a far more efficient and effective approach.

There are many benefits to be gained from the use of static analysis. For example, it scans ALL code. If there are vulnerabilities in the distant corners of a legacy part of an application, then static analysis has a higher probability of finding those vulnerabilities than code review focused on the new works.

Another advantage is that the ability to define and automatically check for the violation of project specific rules provides policing without any manual intervention. If any team member forget to follow those rules, it will be highlighted by static code analysis.

A third major benefit is static code analysis can find violations so early in the development cycle that they can usually be addressed by the developer. Testers are therefore free to concentrate on other matters, and costs are reduced.

In comparing static analysis tools for a project adhering to CERT C, it is important to remember that although the number of CERT C guidelines implemented may provide an obvious comparison statistic, generally it is the easiest ones to include which are common to all of the offerings. Unhappily, the violation checks which are difficult to implement for tool vendors tend to be those which are also tricky to analyse through inspection.

It is also useful to compare the completeness of detection for each of the rules. Many of the guidelines encompass several different possible error cases. Are they all considered? And how extensively is each one dealt with?

Finally, consider that the tools will not be used in isolation but rather as part of a complete development toolchain. Seamless integration into that toolchain will yield rewards in the form of efficient and effective development.

Works Cited

Secure Coding in C and C++ (SEI Series in Software Engineering) 2nd (second) Edition by Seacord, Robert C. published by Addison Wesley (2013)

“SEI CERT C Coding Standard”, Software Engineering Institute, Carnegie Mellon University https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard

“The C Book”, second edition by Mike Banahan, Declan Brady and Mark Doran, originally published by Addison Wesley in 1991. Online Version: http://publications.gbdirect.co.uk/c_book/

www.ldra.com
LDRA

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Inc.
2540 King Arthur Blvd, 3rd Floor, 12th Main Lewisville Texas 75056
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit B-3, Third floor Tower B, Golden Enclave
HAL Airport Road Bengaluru 560017
Tel: +91 80 4080 8707
e-mail: india@ldra.com

Implementing CERT C with the LDRA tool suite® v2.1 06/17