embeddedworld2020
Exhibition&Conference
... it's a smarter world

How to prove that your C/C++ code is safe and secure

Christian Guß
Application Engineering
The MathWorks GmbH
52064 Aachen, Germany

***Abstract*—Are you afraid of finding critical software bugs too late? Would you like to obtain evidence that your code, either self-written or not, is free from overflow, divide-by-zero, out-of-bounds array access, and other run-time errors before you use it in safety- and security-critical systems? Do you need to comply with safety and security standards or guidelines like MISRA, SEI CERT-C, or ISO/IEC TS 17961? In this paper, I discuss sophisticated static analysis methods that verify and prove the absence of run-time errors and vulnerabilities in the source code at unit and integration levels. Utilizing sound formal methods that consider all potential inputs, control- and data flows without code execution, organizations can gain confidence that the software they rely on is safe and secure. This gives organizations more than a mere error detection tool -- it reduces testing and verification costs, and makes code quality transparent across entire teams.**

Keywords—static code analysis, safety, secuity, quality objectives, guidelines, metrics, HIS, MISRA, CERT-C, IEC TS 17961, ISO 26262, functional safety, cybersecurity, dynamic analysis

I. INTRODUCTION

Modern software development is constantly challenged to be fast and low-cost. However, it is notoriously difficult to gain confidence on the safety and security of the produced software in such a setting. This challenge is particularly relevant for high integrity embedded systems, which must reliably perform their intended functionality under all circumstances. Their development process must follow rigorous functional safety standards like IEC 61508 and ISO 26262, which even require to provide *proof* for the absence of errors. It is well known that such proof cannot be obtained by dynamic testing alone, and that companies often overshoot time or cost budgets in an attempt to comply with safety standards. The right static analysis methods and tools effectively address this challenge by providing the needed proofs automatically, and thereby reduce development time and risks.

II. SOPHISTICATED SOFTWARE ANALYIS METHODS

A. *Dynamic Software Execution (Testing)*

The dynamic execution of code is a widely accepted method of quality assurance for software. This is the only method that can be easily applied on the target hardware as well. Dynamic execution of programs is suitable for detecting the presence of errors, but not their absence [1]. This is because (in real applications) the test cases are never exhaustive. Even with full decision, condition and MC/DC coverage, not all possible combinations of input values and parameters are covered, leaving room for subtle defects in numerics, pointers, array accesses and more. Testing is therefore essential, but insufficient to obtain evidence in software quality.

B. *Software Review and Walkthrough*

A software review can uncover additional errors by systematically inspecting the code for patterns and anomalies. The effect of this time-consuming method is highly dependent on the skills and focus of the human reviewer. Especially the implicit rules of the C language can be easily overseen, as shown in code example [Fig. 1].

```
1   unsigned int a = -1;                    -
2   if (a < 0u)                             -
3   {                                       -
```

Fig. 1. Example for c-code that implicitly converts a signed variable into a signed value before comparison with an unsigned value

To compare variable 'a' with the unsigned value '0', the left side operand 'a' is implicitly casted in line 2 into a relatively large unsigned value [Fig. 1]. The human brain is not trained to find patterns like this in huge amounts of unprocessed code, but great in evaluating and classifying already observed abnormalities. Code reviews like this become more effective with computer assisted pre-analyzed code.

C. *Coding Guidelines – Reduction of Language Subset*

Just because the compiler accepts certain language constructs does not mean that it is a good idea to use them. A

very famous example in MISRA C:2012 Rule 15.1 [2] is the usage restriction of the "goto" statement, which makes the program unstructured and difficult to understand. Coding guidelines like MISRA are supposed to help to avoid these doubtful programming techniques. Where finding "goto" statements only requires a text search, it looks a lot more difficult with other rules. The SEI CERT-C rule MSC37-C "Ensure that control never reaches the end of a non-void function" sounds reasonable, because leaving the scope of such a function might return a random value [3]. By further processing this return value, for example in the form of "`if(is_password_ok(void))`", a security vulnerability is created. To detect these deep data and control flow-based problems, sophisticated analysis methods are required.

D. Static Code Analysis

There is a broad range of methods that can be found behind the term static code analysis. The most general description and unique similarity of all is that they do not need to execute the code for verification. Even some compilers give warnings about possible mistakes. For example, the flag "-Wuninitialized" for GNU GCC can catch simple cases of uninitialized variables [4]. Utilizing these warnings makes sense, since every finding can be fixed at an early stage. Early stage also means applying static code analysis before dynamically testing functional requirements, because even a non-initialized variable can cause your code to behave randomly. Testing on random behavior is a common systematic verification mistake which can be recognized by flipping test results without any change.

Although compiler warnings and simple static code analysis tools are helpful, their scope and capabilities are limited. The methods used are mostly based on pattern-matching and heuristics and might not be able to handle the complex control and data flows that occur, for example, during pass-by-reference function calls across multiple compilation units.

The precision of static code analysis can be judged by comparing the truth vs. analysis verdict of each result:

- True-Positive: Detect actual bug in the code.
- False-Negative: Fail to detect bug in the code.
- False-Positive: Healthy code reported as containing bug.
- True-Negative: Healthy code reported as healthy.

An analysis method is "sound" if it never produces False-Negatives. The precision of the results must be measured with different code examples. Results on single file cannot simply be transferred to a complex multi file analysis. A tool for static code analysis must be considered according to these criteria. For example, the MISRA C:2012 guideline contains system-decidable rules that are listed as mandatory [2]. Compliance to this widely accepted guideline therefore requires a tool that provides deep control and data flow analysis at the integrated code and not only on unit level.

E. Formal Verification

Formal verification is a static approach to measure dynamic software quality attributes. It is proving the correctness of atomic operations in the source code regarding to run-time errors [5]. Abstract Interpretation [10] as a formal method use sound approximation of states in computer programs in a more general form. Abstract interpretation thoroughly analyzes all variables of the code.

The substantial computing power required for this analysis has not been readily available in the past. Abstract interpretation, when combined with today's increased processing power, is a practical solution to complex testing challenges. When applied to the detection of run-time errors, abstract interpretation performs a comprehensive verification of all risky operations and automatically diagnoses each operation as proven, failed, unreachable, or unproven. Engineers can use abstract interpretation to obtain results at compilation time, the earliest stage of the testing [6].

III. INDUSTRY USAGE OF FORMAL VERIFICATION

When it comes to proving that software is safe and secure, the use of formal verification is essential. However, the motivation for proving code may vary from company to company.

A. Mapping to functional fastety standards

External motivations are very common when companies decide to use formal methods. Functional safety standards, such as IEC 61508 and ISO 26262-6, recommend or even require the use of formal methods in higher safety integrity levels (SIL). For these companies, compliance with such standards requires the use of formal methods for unit verification.

B. Customer – Supplier relationship

Another significant reason for proving the absence of errors is delivered software. This very often affects supplier relationships in the automotive, aviation and space industries. But also, in-house deliveries and legacy code as well. Delivery requirements are usually defined in advance and can be defined, monitored and reported by using the concept of Software Quality Objectives (SQO) [7]. This makes it possible to define less restrict quality requirements in early project phases than in later phases, such as closely before production.

C. Development process and responsibilities

It is the way formal methods are used during the development process that often determines their effectiveness. Especially when used for compliance with functional safety standards, formal methods are often applied very late in the development process on the whole program code, to prove code correctness. The task is often performed by integration departments and not by the original developers. The main goal in this stage is often to demonstrate compliance towards obtaining final approval, rather than finding and fixing all lingering defects. However, tools for formal verification are designed by their nature to provide a lot of information about the code's nature and potential issues. This information is useful for the developer to fix mistakes and to improve the code design (additional range-checks etc.) but rather overwhelming for quality engineers in a final development stage.

Since the formal verification can prove that software units are free from errors in every integration, the entire set of all software unit proofs is sufficient for the evidence of absence of runtime errors (which are in scope) on the entire integration as well.

For software integrators, a verification tool is suitable which can quickly and efficiently check entire integrations for coding standards and integration defects and compare them with the predefined quality goals.

In summary, different tasks and roles in a company require different approaches and tools. The developer wants to get detailed results on unit level, and the integrator to get compliance results on integration level. For the company it is worthwhile to find a complementary and role-specific tool approach.

IV. Tool Support For Formal Verification

Formal verification is powerful when applied to the right software component and used by the right people in the company. MathWorks has developed Polyspace Code Prover™ [8] for this purpose. The tool uses abstract interpretation as a formal method to prove mathematically sound the absence of critical run-time errors like division by zeros, overflow, array access out of bounds, among others.

For many situations, however, a precise and fast static code analysis tool without mathematical proof is enough. To address this need, MathWorks has developed Polyspace Bug Finder™ [8] as a second complementary tool. The tool also uses the method of abstract interpretation, but trades off soundness for less false positives in the detection of hundreds of defect categories, various coding guidelines from safety and security like CERT-C, CWE, ISO/IEC TS 17961 and code metrics violations.

Both tools Polyspace Bug Finder™ and Polyspace Code Prover™ are intended to be used complementary in the development process. Polyspace Bug Finder™ for the analysis of the entire software for defects, compliance to coding standards and metrics. As well as Polyspace Code Prover™ on critical components as early as possible in the development process to influence design decisions.

A. Compliance and reporting

MathWorks offers a Certification Kit to support tool qualification for common safety standards. Polyspace creates reports that document the code assessment including the review comments from the team.

For easier collaboration in larger teams, Polyspace can be extended with Polyspace Access™ [9] a browser-based code review tool that provides code analysis results from a continuous integration server to the team. This will allow companies to share code quality trends across the team and establish a centralized review workflow across the development cycle.

V. Conclusion And Outlook

Methods and tools to prove that code is safe and secure exist and are used actively by companies during development, especially in the functional safety environment. Nevertheless, the effectiveness depends on the chosen usage. While developers can use the results on a small scope very well to influence design decisions, many tasks, such as finding defects, coding guideline violations and metrics, can be done by static analysis methods with minimized false-positives.

Relying on compiler warnings or simple pattern matching algorithms alone is often not enough. It is essential for companies to define a strategy for using the right method and supporting tools along the development process.

References

[1] Dijkstra, “Notes On Structured Programming”, 1972

[2] Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013

[3] SEI CERT. C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. 2016

[4] Using the GNU Compiler Collection (GCC), <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

[5] Wissing K., “Static Analysis of Dynam–c Properties - Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors”

[6] Code Verification and Run-Time Error Detection Through Abstract Interpretation. White paper, MathWorks, www.polyspace.com, 2019

[7] Software Quality Objectives for Source Code v.3.0, <http://www.mathworks.com/tagteam/72337_Software_Quality_Objectives_V3.0.pdf>

[8] Polyspace Product Page, MathWorks, <http://de.mathworks.com/products/polyspace/>

[9] Polyspace Access Product Page, MathWorks, <https://www.mathworks.com/products/polyspace-bug-finder.html#access>

[10] Cousot & Cousot, “Abstract Interpretation”, Proc. Symposium on Principles of programming languages, Los Angeles USA, 1977