ADELARD
part of ncc group

Experience with static analysis – looking for needles in haystacks

Robert Stroud

Research presentation
IFP WG 10.4 Winter Meeting

6 May 2026

Together we're creating a more secure digital future

Motivation

Looking for needles in haystacks

The idea behind this talk is to provide some insight into the kind of issues we encounter when we apply static analysis tools to the software component of safety-critical systems that have already been certified to a high level of safety integrity.

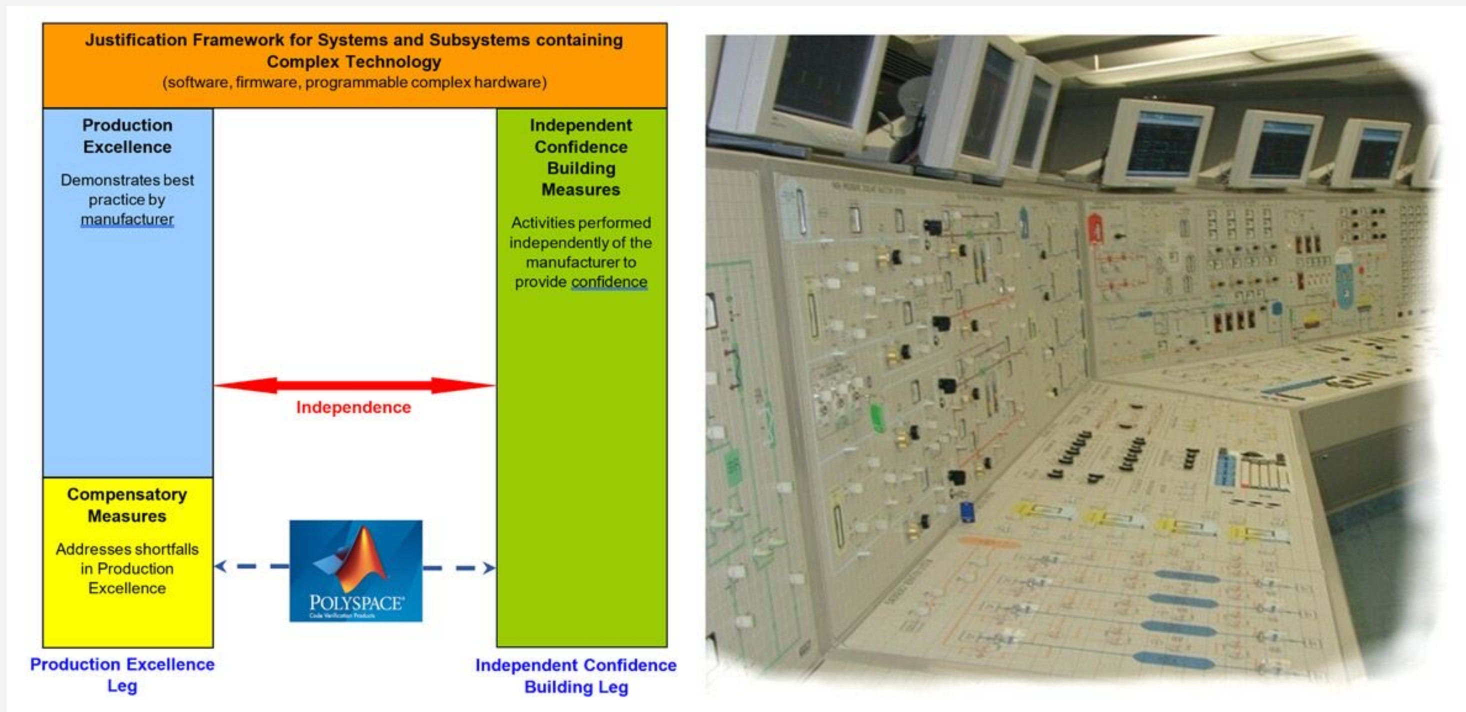
For example, for a SIL 2 system in continuous operation, the probability of a dangerous failure per hour is expected to be between 10^{-6} and 10^{-7}.

In the worst case, you would expect it fail once every 1,000,000 hours, which is roughly every 100 years (876,600 hours)

Bugs that occur once every 100 years are unlikely to be found during testing, so more formal methods of analysis are required.

Motivation

EDF assesses nuclear plant safety with formal methods using Polyspace



“In a recent project, the EDF team demonstrated a very low defect density of 0.07 defects per 1,000 lines of code”
https://uk.mathworks.com/company/user_stories/edf-enhances-nuclear-plant-software-safety-with-polyspace.html

Motivation

How good is 0.07?

"There is a general consensus in some areas of the safety critical systems community that a fault density of about 1 per thousand Lines of Code (kLoC) is world class. Some software, e.g., that for the Space Shuttle, is rather better but fault densities of lower than 0.1 per kLoC are exceptional."

McDermid and Kelly [2006]

What is static analysis?

How good is static analysis?

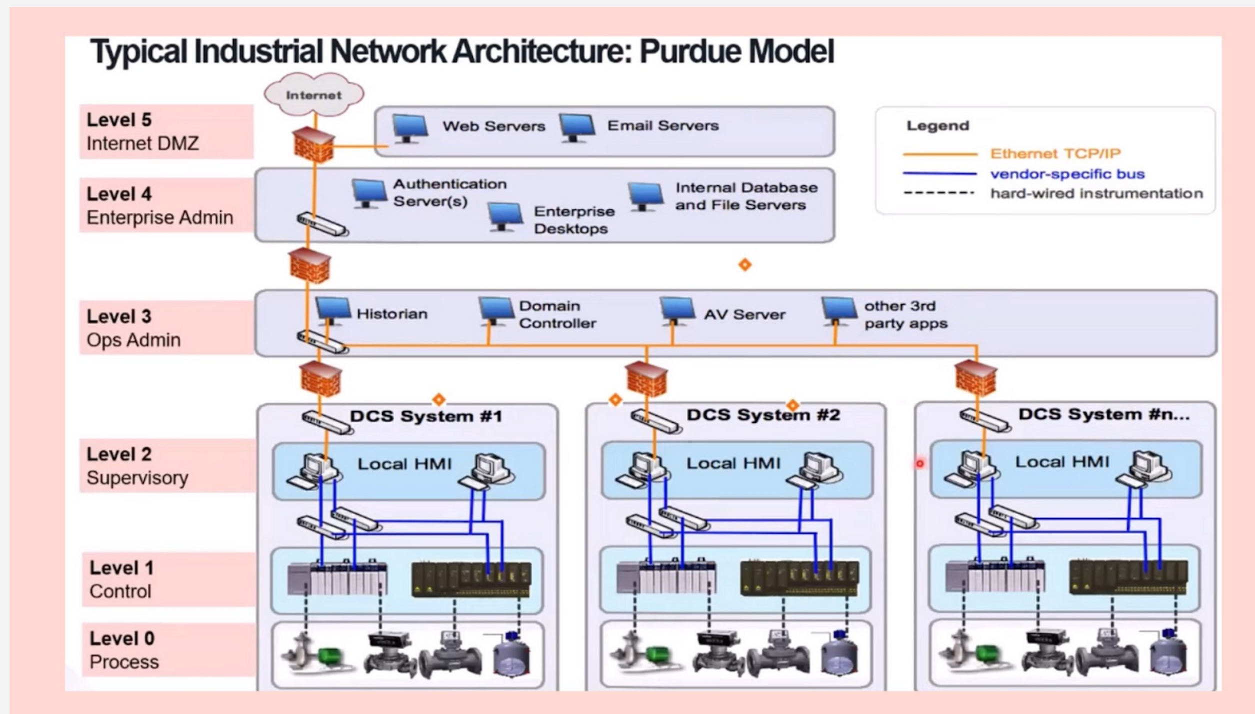
“Using state-of-the-art static analysis (PolySpace, CodeSonar, Astrée, Frama-C, Coverity, etc.) in combination with strict coding standards and full safety-critical development processes, the best achievable defect density is on the order of ~0.1 defects/KLOC, which rivals the numbers reported from formal methods in small systems — but without the absolute guarantees of formal proof.”

ChatGPT, September 2025

adelard.com ADELARD part of ncc group

Case study

Process automation system



https://securityboulevard.com/2023/11/a-guide-to-purdue-model-for-ics-security/

Case study

Analysis of I/O modules for a process automation system

Mature system, widely used

- Originally written in assembly
- Reverse engineered to C

Analysed using Polyspace Bug Finder

Configured to detect 8 kinds of defect

Initial results

- 2462 findings
- 165,785 lines of code
- 14.85 defects / KLOC

Category	Examples
Numerical	Division by zero, type conversion, negative shift operations
Static Memory	Out of bounds arrays, null pointers, use of standard memory, and string libraries
Dynamic Memory	Freed pointers, memory leaks, and unprotected memory
Programming	Assignment versus equality operators, type mismatch, wraparound, and string arrays
Data Flow	Unreachable code, uninitialised variables, and write not followed by a read
Concurrency	Data race
Resource Management	Unclosed file stream or use of a closed file stream
Good Practice	Hard-coded memory buffer size, unused function parameters

adelard.com ADELARD part of ncc group

Case study

Analysis of I/O modules for a process automation system

Mature system, widely used
- Originally written in assembly
- Reverse engineered to C

Analysed using Polyspace Bug Finder

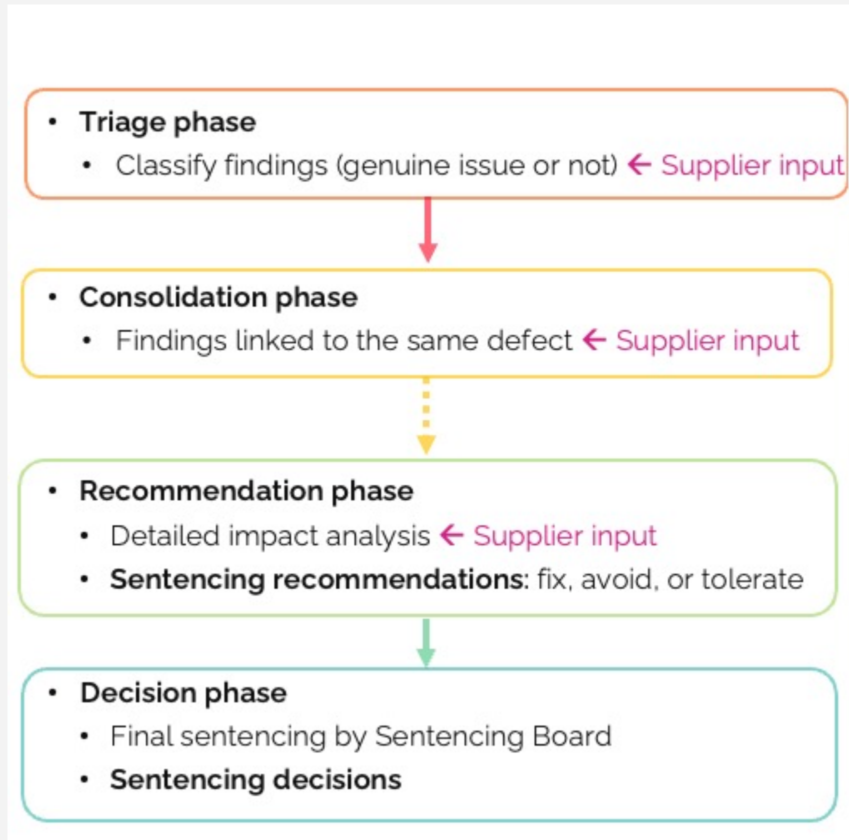
Configured to detect 8 kinds of defect

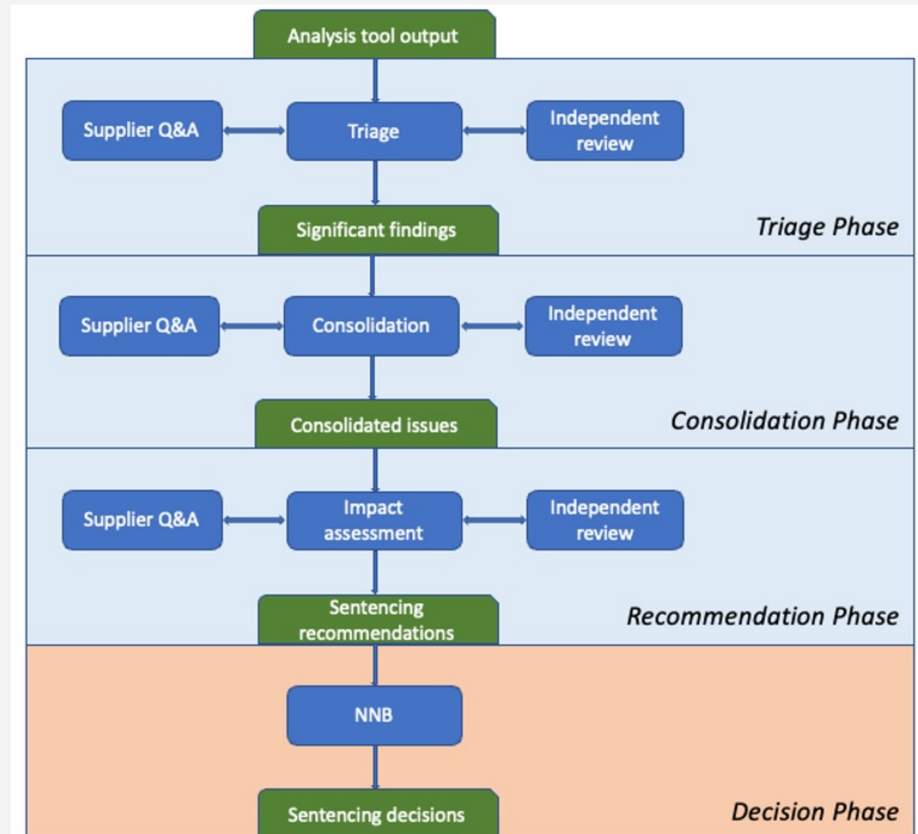
Final results (after sentencing)
- 8 significant findings
- 120,499 lines of code
- 0.07 defects / KLOC

Category	Examples
Numerical	Division by zero, type conversion, negative shift operations
Static Memory	Out of bounds arrays, null pointers, use of standard memory, and string libraries
Dynamic Memory	Freed pointers, memory leaks, and unprotected memory
Programming	Assignment versus equality operators, type mismatch, wraparound, and string arrays
Data Flow	Unreachable code, uninitialised variables, and write not followed by a read
Concurrency	Data race
Resource Management	Unclosed file stream or use of a closed file stream
Good Practice	Hard-coded memory buffer size, unused function parameters

Sentencing the findings

Sentencing process





adelard.com ADELARD part of ncc group

Case study

Results of sentencing process

Phase	Output	No of issues
Analysis	Analysis tool output	2462
Triage	Significant findings	95
Consolidation	Consolidated issues	13
Recommendation	Sentencing recommendations	8

Analysis of results

Significant findings

ID	Issue
1	Race condition affecting the Level 2 diagnostics
2	Race condition affecting timing measurements
3	Race condition affecting timestamp on Level 2 diagnostics
4	Invalid diagnostic requests can return arbitrary data
5	Sync counter can overflow
6	Race condition affecting redundant configuration
7	Race condition affecting redundant configuration
8	Race condition affecting timestamp of Level 2 diagnostics

Example – timer defect

Source code

```
if ( !started )
{
   // start timer
   start = clock;
   started = true;
}
else if ( start + duration < clock )
{
   // do something
}
else
{
   // stop timer
   started = false;
}
```

Example – timer defect

Source code

```
if ( !started )
{
   // start timer
   start = clock;         // non-atomic - 16-bit word, clock = 32 bits
   started = true;
}
else if ( start + duration < clock )
{
   // do something
}
else
{
   // stop timer
   started = false;
}
```


adelard.com ADELARD part of ncc group

Example – timer defect

Clock race condition

Action	Start	Clock
Initial state	0x0000 0000	0x0000 FFFF
Start[low] = Clock[low]	0x0000 FFFF	0x0000 FFFF
Clock++		0x0001 0000
Start[high] = Clock[high]	0x0001 FFFF	0x0001 0000

adelard.com ADELARD part of ncc group

Example – timer defect

Clock race condition – how often does it occur?

A 16-bit 1 ms clock wraps round every 65,536 ms

The instruction time is 100 ns

So the probability that the clock is in a partially updated state is

$100 / (65{,}536 \times 10^{6}) = 1.526 \times 10^{-9}$

If the clock is read once per second, the interval between race conditions is

$1 / (1.526 \times 10^{-9}) = 6.5536 \times 10^{8}$ seconds

This works out at roughly once every 20.8 years, so this is a 20-year bug

But if you have 60 of these devices, it’s a 4-month bug…

Summary

Conclusions

Modern static analysis tools have successfully industrialised formal methods

- Tools are designed to be used by programmers, not mathematicians

At a superficial level, they can be used to enforce coding style rules and check for common programming errors

At a deeper level, they can used to detect obscure concurrency bugs and prove the absence of certain types of errors and vulnerabilities in software

The successful application of static analysis techniques provides strong guarantees that software-related risks in safety systems have been minimized

However, the process is by no means automatic - the initial findings need to be carefully reviewed and sentenced manually, which requires knowledge and skill from the analyst

Analysing third-party code is complicated by the lack of domain knowedge and requires input from the supplier and good communication and comprehension skills from the analyst

Thank you.

Together we're creating a more secure digital future

© 2025 NCC Group. All rights reserved.
Please see www.nccgroup.com for further details. No reproduction is permitted in whole or part without written permission of NCC Group.
This content is for general purposes only and should not be used as a substitute for consultation with professional advisors.