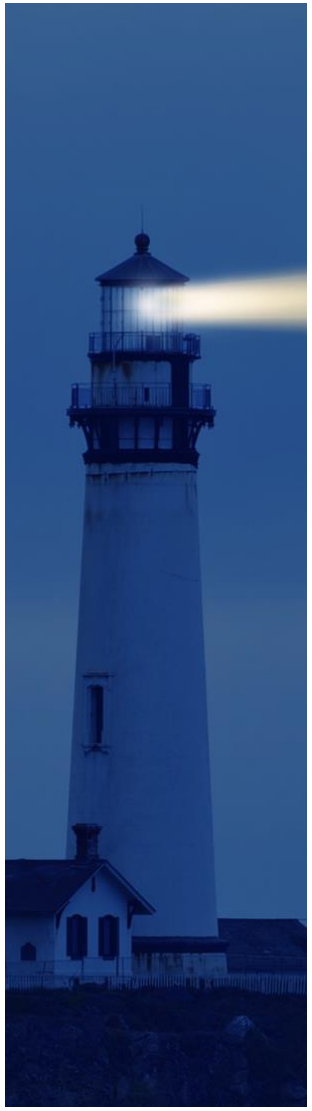
Software Integrity for Safety-Critical Applications

Dr. Daniel Kästner
AbsInt GmbH
2025

AbsInt

AbsInt Key Products



aiT WCET Analyzer

- Safe upper bounds on worst-case execution time: timing guarantees

TimingProfiler

- Static worst-case execution time estimates: continuous timing feedback

TimeWeaver

- Hybrid worst-case timing analysis: combines tracing with static analysis

StackAnalyzer

- Safe upper bounds on maximal stack: no more stack overflows

CompCert

- Formally verified optimizing C compiler: unparalleled level of confidence

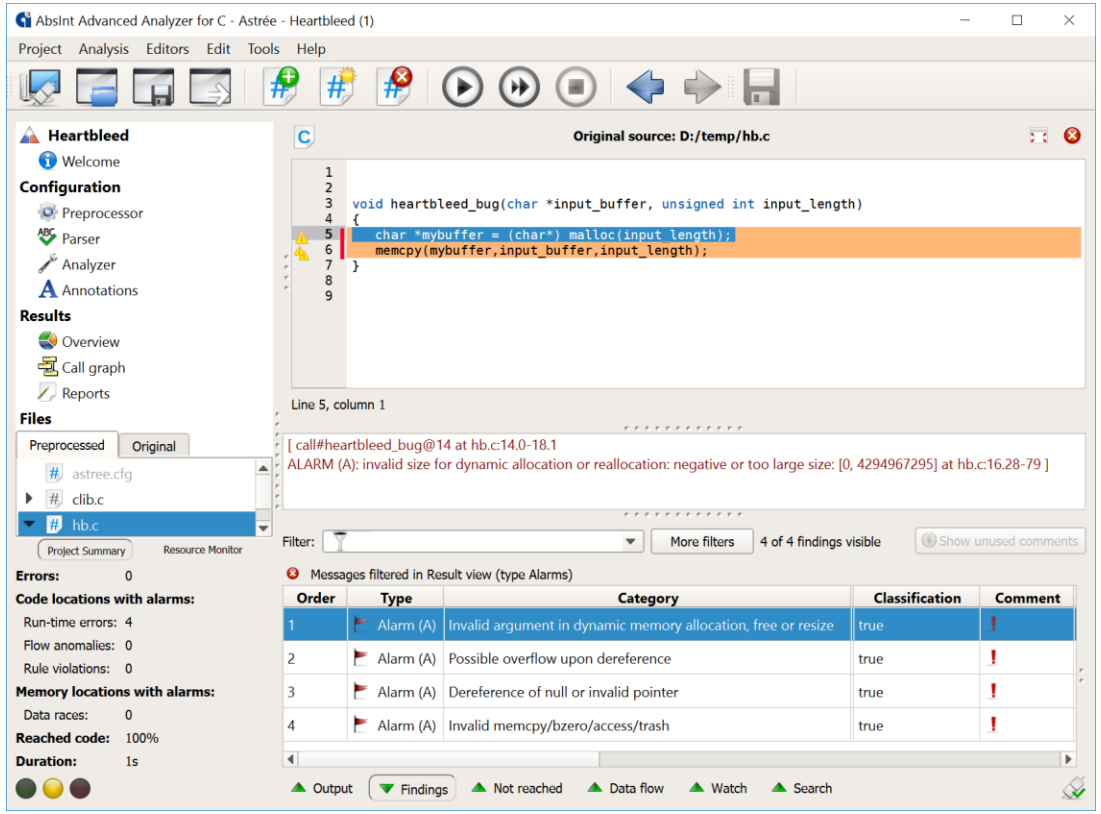
ACM Software System Award 2021

Astrée

- Detects all runtime errors, data races, deadlocks, and other critical errors
- Checks coding guidelines (MISRA, CERT, ...)

A Security Issue ?

```
void heartbleed_bug(char *input_buffer, unsigned int input_length) {
    char *mybuffer = (char*) malloc(input_length);
    memcpy(mybuffer,input_buffer,input_length);
}
```



- Heartbleed bug (2014)
- Security bug in OpenSSL
- Passwords, social insurance numbers, patient records, … leaked
- Millions of people affected
- Estimated cost >$500M*

- Underlying code defects are safety-relevant!
- Defects detectable by static analysis

*https://en.wikipedia.org/wiki/Heartbleed (retrieved April 2017)

Dependability

- Functional Safety
  - Absence of unreasonable risk to life and property caused by malfunctioning behavior of the system
- Security
  - Absence of harm caused by malicious (mis-)usage of the system
- Reliability:
  - Probability with which the system performs its required functions under specified conditions for a specified period of time
- Availability:
  - Probability with which the system operates at a random time within its life range

(Information-/Cyber-) Security Aspects

- Confidentiality
  - Information shall not be disclosed to unauthorized entities
  - ⇨ safety-relevant
- Integrity
  - Data shall not be modified in an unauthorized or undetected way
  - ⇨ safety-relevant
- Availability
  - Data is accessible and usable upon demand
  - ⇨ safety-relevant

\+ Safety

In some cases: not safe ⇒ not secure
In some cases: not secure ⇒ not safe

of the solution. More importantly, fail secure should not impact airworthiness, i.e. fail safe takes precedence over fail secure.

DO-356A. Airworthiness Security Methods and Considerations, 2018.

Memory Safety

There are two broad categories of memory safety vulnerabilities: spatial and temporal. Spatial memory safety issues result from memory accesses performed outside of the "correct" bounds established for variables and objects in memory. Temporal memory safety issues arise when memory is accessed outside of time or state, such as accessing object data after the object is freed or when memory accesses are unexpectedly interleaved.[vi] Many of the major cybersecurity vulnerabilities over the past several decades were facilitated by memory safety vulnerabilities, including the Morris Worm of 1988, the Slammer Worm denial-of-service attack in 2003, the Heartbleed vulnerability in 2014, and the BLASTPASS exploit chain of 2023.[vii] For over 35 years, this same class of vulnerability has vexed the digital ecosystem.

to eliminate the most prevalent classes. Given the complexities of code, testing is a necessary but insufficient step in the development process to fully reduce vulnerabilities at scale. If correctness is defined as the ability of a piece of software to meet a specific security requirement, then it is possible to demonstrate correctness using mathematical techniques called *formal methods*. These techniques, often used to prove a range of software outcomes, can also be used in a cybersecurity context and are viable even in complex environments like space. While formal methods have been

While there are several types of formal methods that span a range of techniques and stages in the software development process, this report highlights a few specific examples. *Sound static analysis* examines the software for specific properties without executing the code.[xxi] This method is effective because it can be used across many representations of software, including the source code, architecture, requirements, and executables. *Model checkers* can answer questions about a number

Back to the Building Blocks: A Path Toward Secure and Measurable Software. The White House, USA. 2024.

Unforgivable Defects

Malicious Cyber Actors Use Buffer Overflow Vulnerabilities to Compromise Software

Note: This Secure by Design Alert is part of an ongoing series aimed at advancing industry-wide best practices to eliminate entire classes of vulnerabilities during the design and development phases of the product lifecycle. The Secure by Design initiative seeks to foster a cultural shift across the technology industry, normalizing the development of products that are secure to use out of the box. Visit cisa.gov to learn more about the principles of Secure by Design, take the Secure by Design Pledge, and stay informed on the latest Secure by Design Alerts.

The software development community has twenty years of extensive knowledge and effective solutions for buffer overflows—however, many software manufacturers continue to expose customers to products with these vulnerabilities.

Buffer overflow vulnerabilities are a prevalent type of memory safety software design defect that regularly lead to system compromise. The Cybersecurity and Infrastructure Security Agency (CISA) and Federal Bureau of Investigation (FBI) recognize that memory safety vulnerabilities encompass a wide range of issues—many of which require significant time and effort to properly resolve. While all types of memory safety vulnerabilities can be prevented by using memory safe languages during development, other mitigations may only address certain types of memory safety vulnerabilities. Regardless, buffer overflow vulnerabilities are a well understood subset of memory safety vulnerability and can be addressed by using memory safe languages and other proven techniques listed in this Alert. Despite the existence of well-documented, effective mitigations for buffer overflow vulnerabilities, many manufacturers continue to use unsafe software development practices that allow these vulnerabilities to persist. For these reasons—as well as the damage exploitation of these defects can cause—CISA, FBI, and others[1] designate buffer overflow vulnerabilities as unforgivable defects.

EU Cyber Resiliance Act CRA

- Comes into force end of 2027
- Introduces mandatory cybersecurity requirements for manufacturers and retailers, governing the planning, design, development, and maintenance of all products that contain a digital component (HW or SW) where the intended purpose includes connectivity (incl. file I/O).
- Excluded are certain open-source software or services, and products that are already covered by existing rules, e.g. medical devices, aviation and cars.
- Note: the regulation does apply to products *using* open source software.

Official Journal
of the European Union

EN
L series

2024/2847

20.11.2024

REGULATION (EU) 2024/2847 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL

of 23 October 2024

on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) No 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act)

3.1.2 (2) Products with digital elements shall be delivered without any known exploitable vulnerabilities;

Sample Sub-requirements:

— A vulnerability assessment should be performed against the digital elements of a product

— Known exploitable vulnerabilities shall be fixed before the release of the product

Keywords: exploitable vulnerability, vulnerability assessment

Functional Safety

- Demonstration of functional correctness
  - Functional requirements are satisfied
  - ➢ Automated and/or model-based testing
  - ➢ Formal techniques: model checking, theorem proving
- Satisfaction of safety-relevant quality requirements
  - No runtime errors (e.g. division by zero, overflow, invalid pointer access, out-of-bounds array access)
  - Resource usage:
    - Timing requirements (e.g. WCET, WCRT)
    - Memory requirements (e.g. no stack overflow)
  - Robustness / freedom of interference (e.g. no corruption of content, incorrect synchronization, illegal read/write accesses)
  - Compliance with the software architecture, data and control coupling
  - ➢ Insufficient: Tests & Measurements
    - No specific test cases, unclear test end criteria, no full coverage possible
  - ➢ **Static analysis**
    - ⇨ Formal technique (sound): Abstract Interpretation – no defect missed

REQUIRED BY
DO-178B / DO-178C / ISO-26262, EN-50128, IEC-61508

REQUIRED BY
DO-178B / DO-178C / ISO-26262, EN-50128, IEC-61508

+ Security-relevant
ISO 21434, DO-356A, …

- Code Guideline Checking
- Runtime Error Analysis / Data & Control Flow Analysis / Data and Control Coupling
- Code Metrics
- WCET Analysis
- Stack Usage Analysis

Trends in Safety-Critical Systems

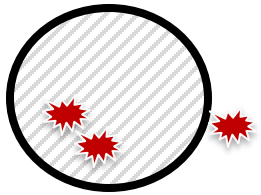
- Increasing connectivity in safety-critical systems
  - Cloud-based services
  - C2X communication
  - Over-the-air updates
  - Smart mobility / grid / …
  - ⇨ Increasing frequency and attack scale of cybersecurity issues
    - 2015 FDA blacklisting Hospira Symbiq infusion pump (Wifi tampering)
    - 2015 General Motors OnStar RemoteLink App
    - 2016 Jeep Cherokee hack (Fiat Chrysler Uconnect)
    - 2017 CAN Bus Standard Vulnerability (ICS-ALERT-17-209-01)
- Autonomy – Increasing complexity and criticality
  - Highly automatic driving, Unmanned Aerial Vehicles, robotics, smart medical devices, …
- Growing data cloud
  - medical data, movement tracking, financial data, social media meta data, …
  - ⇨ Privacy concerns more critical

⇨ Increasing risk of critical software defects

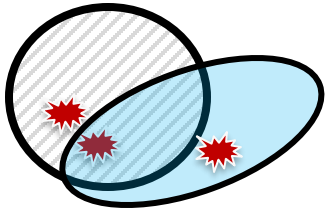
Static Program Analysis

- Computes results only from program structure, without executing the software.
- Categories, depending on analysis depth:
  - Syntax-based: Coding guideline checkers (e.g. MISRA C)
  - Semantics-based

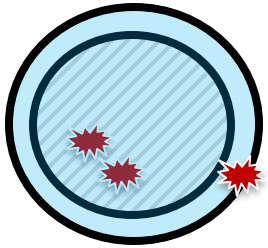
Question: Is there an error in the program?
- False positive: answer wrongly "Yes"
- False negative: answer wrongly "No"

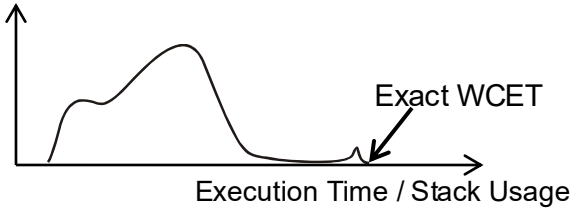


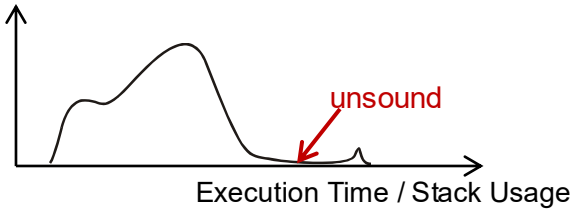
- Unsound: Bug-finders / bug-hunters.
  - False positives: possible
  - False negatives: possible



☑Sound / Abstract Interpretation-based
  - False positives: possible
  - No false negatives ⇨ Soundness
    No defect missed







Analysis Depth

- Division by zero
  - `a/0` → division by zero always happens
    - can be detected syntactically
  - `a/b` → division by zero can occur if b might be zero
    - semantic information needed: value range of b

  - Unsound analyzer:
    - Alarm on `a/b`: division by zero might happen
    - No alarm on division on `a/b`: division by zero might still happen!

  - Sound analyzer:
    - Alarm on `a/b`: division by zero might happen
    - No alarm on division on `a/b`: proof that $b \neq 0$, no division by 0 possible

ISO 26262: Resource Constraints

7.4.13 An upper estimation of required resources for the embedded software shall be made, including:

a) the execution time;

- Worst-case execution time (WCET)
- Worst-case response time (WCRT) ≈ WCET + preemption time + blocking time

b) the storage space; and

EXAMPLE RAM for stacks and heaps, ROM for programme and non-volatile data.

- Worst-case stack usage

Excerpt from:
ISO 26262-6 Road vehicles - Functional safety – Part 6: Product development: Software Level, 2018.

ISO 26262 – SW Unit Design and Implementation

8.4.5 Design principles for software unit design and implementation at the source code level as listed in Table 6 shall be applied to achieve the following properties:

a) correct order of execution of sub-programmes and functions within the software units, based on the software architectural design;

b) consistency of the interfaces between the software units;

c) correctness of data flow and control flow between and within the software units;

d) simplicity;

e) readability and comprehensibility;

f) robustness;

EXAMPLE Methods to prevent implausible values, execution errors, division by zero, and errors in the data flow and control flow.

g) suitability for software modification; and

h) verifiability.

Excerpt from:
ISO 26262-6 Road vehicles - Functional safety – Part 6: Product development: Software Level, 2018.

ISO 26262 – Software Unit Verification

Table 7 — Methods for software unit verification

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Walk-through[a]	++	+	o	o
1b	Pair-programming[a]	+	+	+	+
1c	Inspection[a]	+	++	++	++
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[b, c]	+	+	++	++
1g	Data flow analysis[b, c]	+	+	++	++
1h	Static code analysis[d]	++	++	++	++
1i	Static analyses based on abstract interpretation[e]	+	+	+	+
1j	Requirements-based test[f]	++	++	++	++
1k	Interface test[g]	++	++	++	++
1l	Fault injection test[h]	+	+	+	++
1m	Resource usage evaluation[i]	+	+	+	++
1n	Back-to-back comparison test between model and code, if applicable[j]	+	+	++	++

[a] For model-based development these methods are applied at the model level, if evidence is available that justifies confidence in the code generator used.

[b] Methods 1f and 1g can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1f and 1g can be part of methods 1e, 1h or 1i.

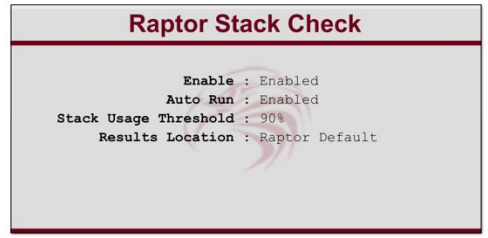
[d] Static analyses are a collective term which includes analysis such as searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines.

[e] Static analyses based on abstract interpretation are a collective term for extended static analysis which includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation.

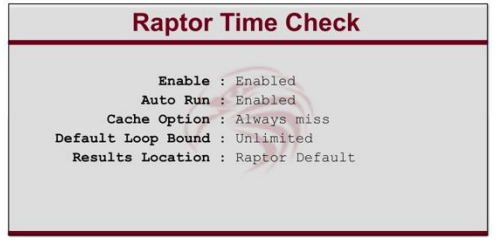
Excerpt from:
ISO 26262-6 Road vehicles - Functional safety – Part 6: Product development: Software Level, 2018.

Covered V&V Activities

- Worst-case stack usage analysis
  - ⇨ StackAnalyzer / Raptor Stack Check

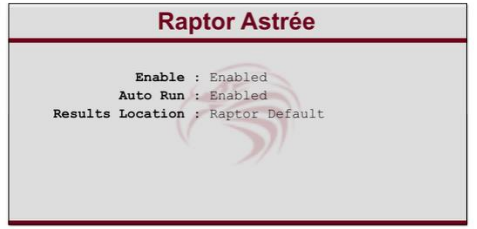


- Worst-case execution time estimation
  - ⇨ TimingProfiler / Raptor Time Check



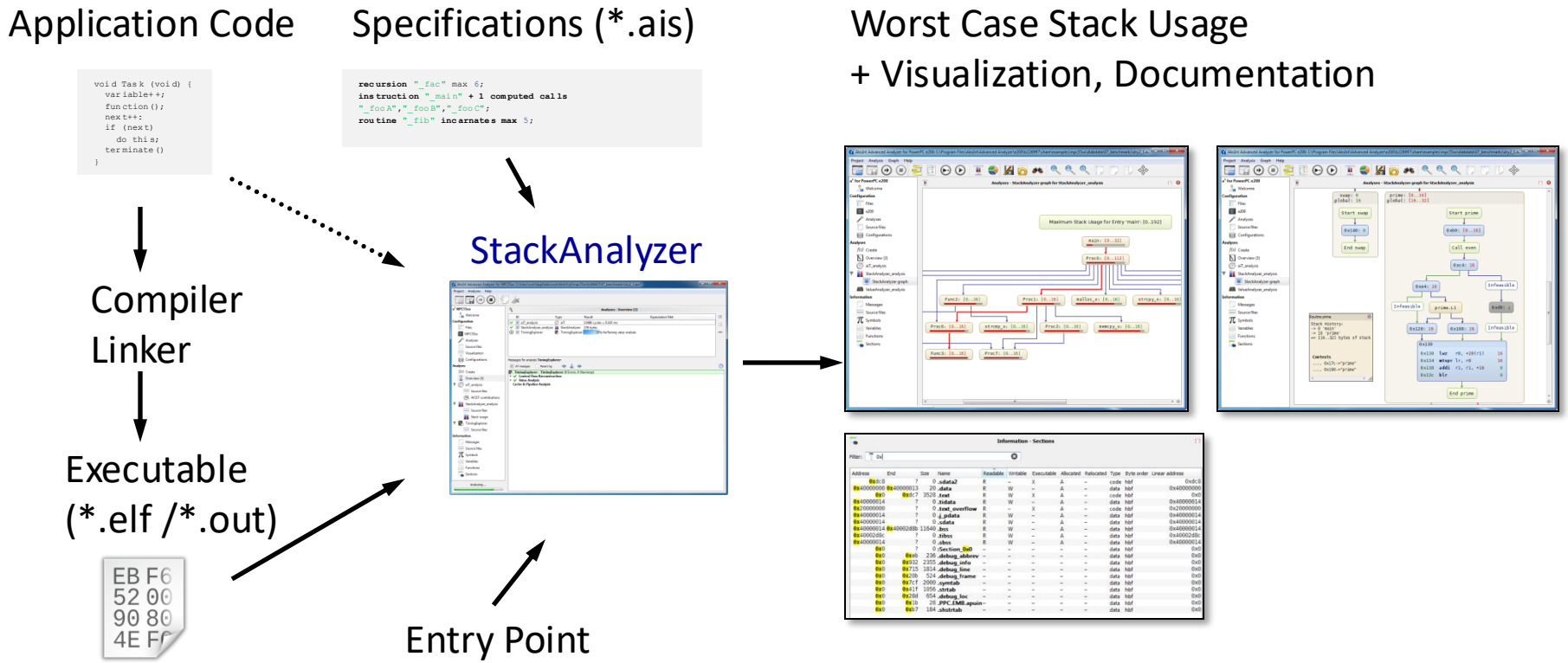
- Worst-case execution time analysis
  - ⇨ aiT WCET Analyzer, TimeWeaver

- Runtime error analysis, code guideline checking & vulnerability scanning
  - ⇨ Astrée / Raptor Astrée



StackAnalyzer

- Abstract Interpretation-based static analysis at binary code level
- StackAnalyzer computes safe upper bounds of the stack usage of the tasks in a program for all inputs



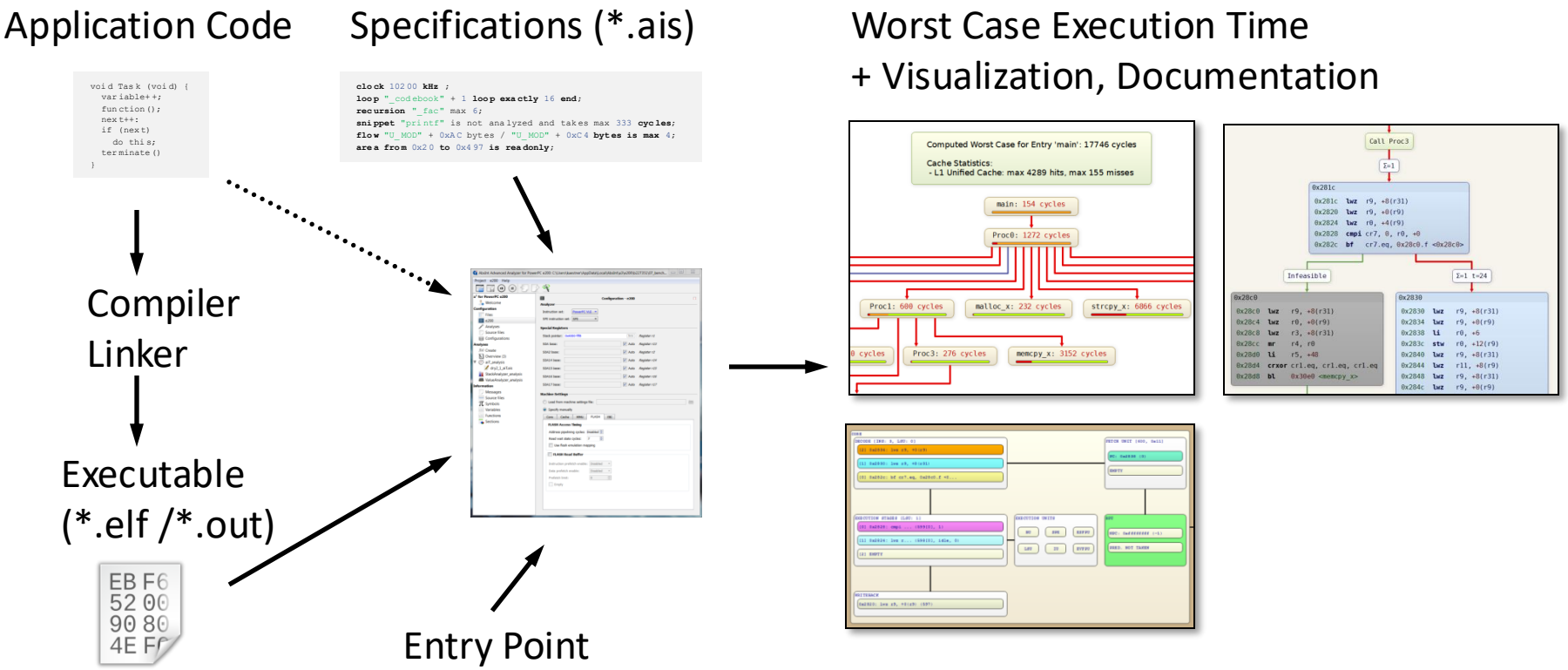
Example Targets:

Am486	ARM
C16x & ST10	C28x
C33x	dsPIC
ERC32	FR81S
HCS12	i386DX
LEON2	LEON3 & LEON4
M68k & ColdFire	MCS251
MCS51	MIPS32
MSP430(x)	Nios II
PowerPC	Renesas RX
RISC-V	New: RL78
S12Z	SuperH
TriCore	V850 & RH850
x86	ARC

aiT Worst-Case Execution Time Analyzer

- Global static program analysis by Abstract Interpretation: microarchitecture analysis (caches, pipelines, …) + value analysis + path analysis

⇨ Safe and precise WCET bounds for timing-predictable processors



Example Targets:
PowerPC
55xx/56xx/57xx/…
TriCore/AURIX
ARM Cortex
M0/M3/M4/R4F/R5F…
C28x, LEON, V850, …
RISC-V*

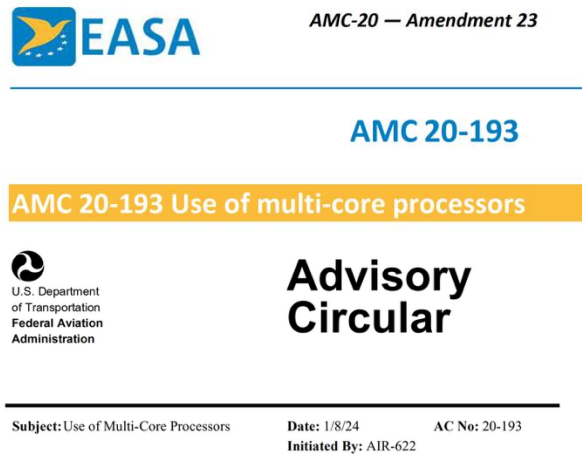
Multi-Core Concerns

- Contention on shared resources:
  - Memory, caches, external interfaces, prefetch buffers, …
- Typically arbitrated by interconnects / coherency fabric
  - Accesses are subject to delays
  - Maximal delay undocumented and/or unbounded?
  - Transactions might be reordered

⇨ Interference channels have to be identified and mitigated

- ⇨ Time interference
- ⇨ Resource interference

⇨ Either robust time and resource partitioning, or WCET has to be determined with all software components on all cores executing in the intended final configuration [CAST32A, AMC20-193]


EASA

AMC-20 — Amendment 23

AMC 20-193

AMC 20-193 Use of multi-core processors

U.S. Department of Transportation
Federal Aviation Administration

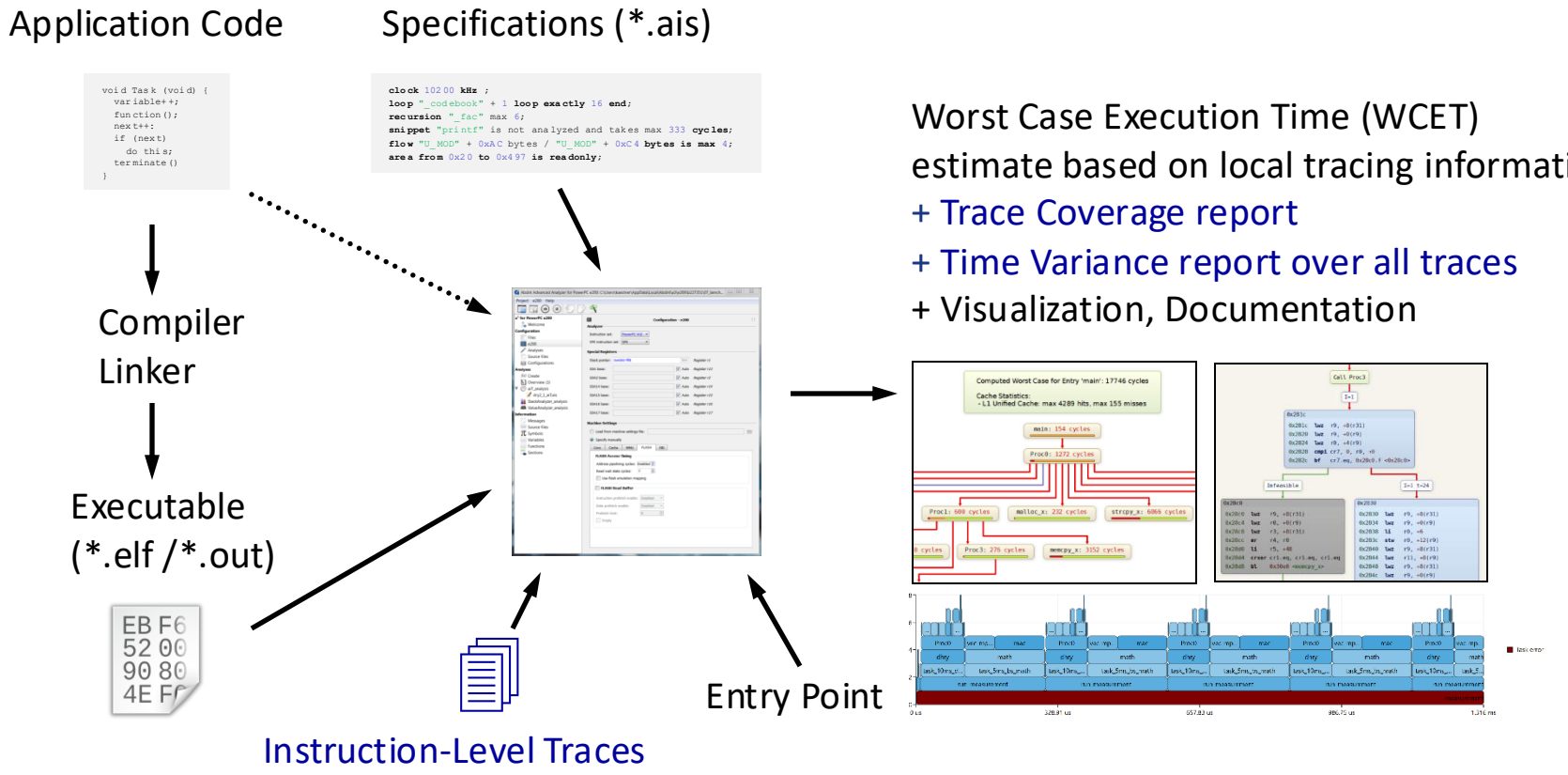
Advisory Circular

Subject: Use of Multi-Core Processors Date: 1/8/24 AC No: 20-193
Initiated By: AIR-622

[AMC 20-193] EASA. AMC-20 – Amendment 23. AMC 20-193. Use of multi-core processors, 2022.
[AC20-193] FAA. Advisory Circular AC No 20-193. Use of Multi-Core Processors, 2024.

TimeWeaver – Non-intrusive Hybrid WCET

- Focus: non-timing-predictable microprocessors
- Combines static value and worst-case path analysis and hardware measurements based on non-intrusive instruction-level tracing

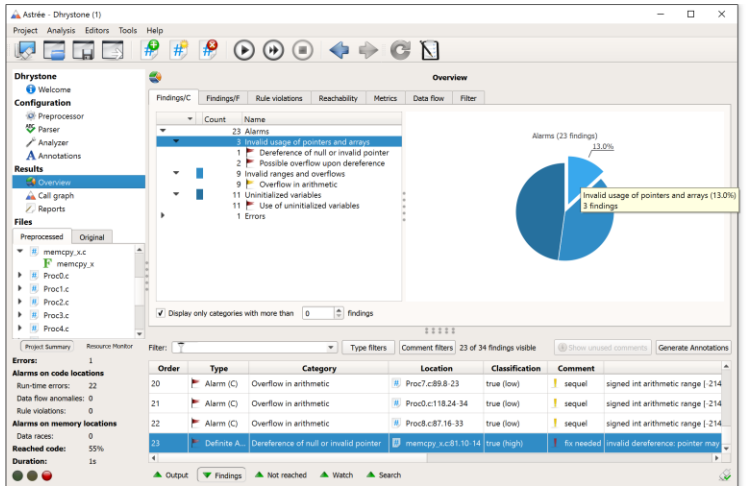


Example Targets:
ARM Cortex R5F/R52/
M7/A53/A72/AXX…
PowerPC QorIQ
P40xx/P50xx…
RISC-V*

Also applicable to:
TriCore/AURIX

Astrée

- Abstract Interpretation-based static runtime error analysis at source code level
- Astrée detects all runtime errors* with few false alarms:
  - Covered defect classes: array index out of bounds, int/float division by 0, invalid pointer dereferences, uninitialized variables, arithmetic overflows, data races, lock/unlock problems, deadlocks, …
  - Ensures C/C++-level memory safety
  - Data and control flow analysis, data and control coupling analysis
  - Non-interference analysis, signal flow analysis, taint analysis for data safety, security and fault propagation, SPECTRE detection
  - \+ User-defined assertions, unreachable code, non-terminating loops, alias analysis
  - \+ Check coding guidelines (MISRA C/C++, Adaptive AUTOSAR C++, CERT C/C++, CWE, ISO TS 17961)
  - \+ Automatic support for ARINC653/OSEK/AUTOSAR OS configurations

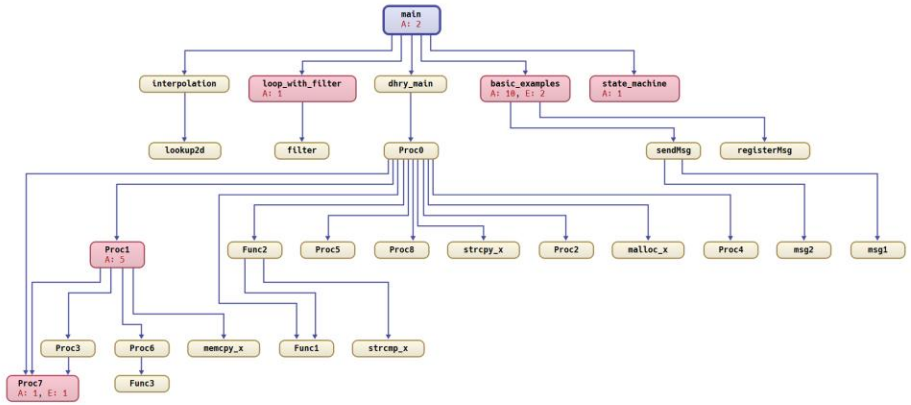


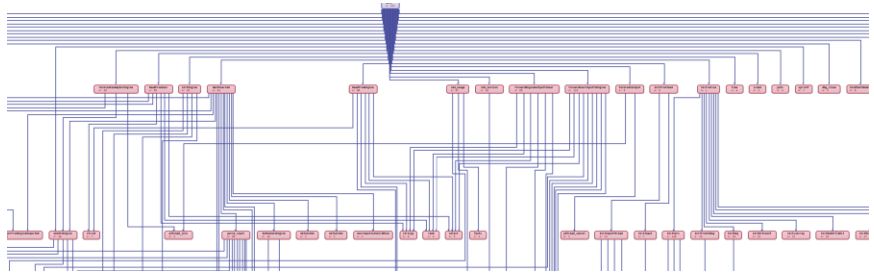
RuleChecker included

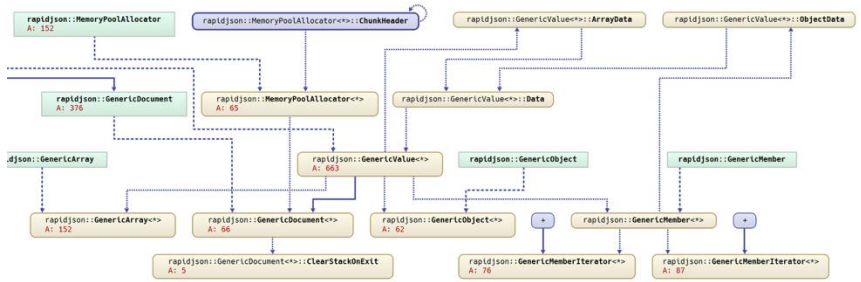
* Defects due to undefined / unspecified behaviors of the programming language

Data and Control Flow Analysis in Astrée

- Control flow analysis
  - Caller/callee relationships between functions
  - Call graph
  - Function calls per concurrent thread
- Data flow analysis
  - List of global/static variables with information about
    - locations/functions/processes performing read/write accesses
    - access properties:
      - Thread-local
      - Shared
      - Subject to data race
- Data and control coupling / Interference analysis
- Soundness: no data/control flow is missed
  - Aware of data and function pointers, task interference, …
  - Freedom of interference can be proven



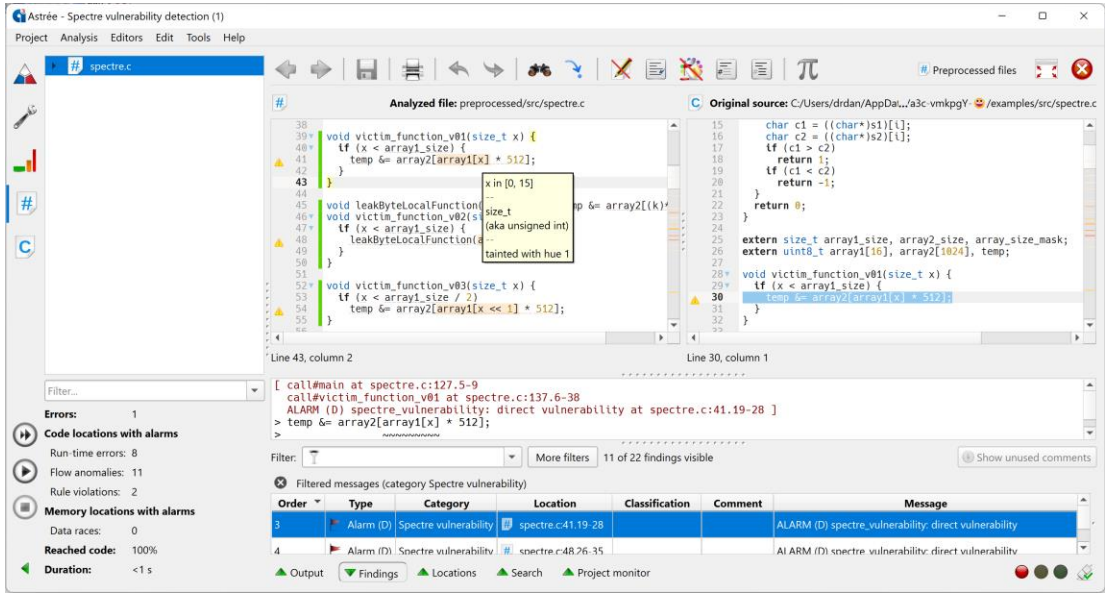




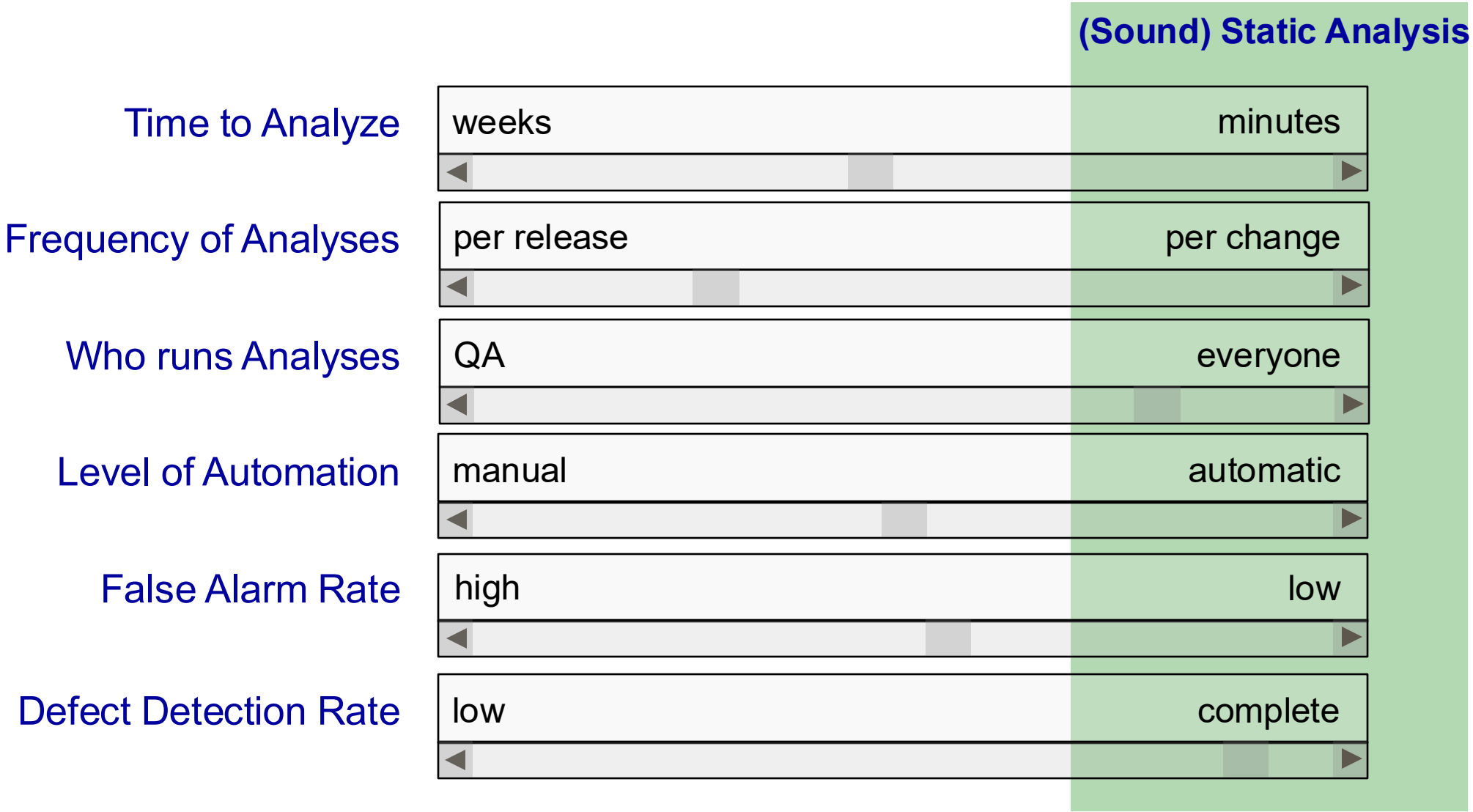
Support for Cybersecurity Analysis

- Many security vulnerabilities due to undefined / unspecified behaviors in the programming language semantics:
  - buffer overflows, invalid pointer accesses, uninitialized memory accesses, data races, etc.
  - Consequences: denial-of-service, code injection, data breach
  - ⇨ Absence can be proven by Astrée

- Astrée provides further sophisticated analysis modules:
  - Checking coding guidelines (RuleChecker included)
  - Data and Control Flow Analysis
  - Taint Analysis
    - Impact analysis (data safety / “fault” propagation)
    - Non-interference analysis (signal flow analysis, freedom of interference)
  - Side channel attacks
    - SPECTRE detection (Spectre V1/V1.1, SplitSpectre)



Efficiency


(Sound) Static Analysis
Time to Analyze
weeks
minutes
Frequency of Analyses
per release
per change
Who runs Analyses
QA
everyone
Level of Automation
manual
automatic
False Alarm Rate
high
low
Defect Detection Rate
low
complete

Can You Trust Your Compiler?

- Compilers are complex software
- A buggy compiler can introduce bugs in compiled code: miscompilation
- Compiler bugs can cause erroneous and erratic behavior
- Compiler bugs are difficult to identify
- Miscompilation invalidates safety guarantees provided by source-level verification

The CompCert Compiler

- Formally verified optimizing C Compiler
  - CompCert avoids miscompilation by formal verification of the compiler itself
  - The code it produces is proved to behave exactly as specified by the semantics of the source C program.
  - This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest software assurance levels.
- Validates source-code verification activities
  - All safety properties verified on the source code automatically hold for the generated code as well.
- Enables proven traceability between source code and executable object code
- The performance of the generated code is comparable to state-of-the-art modern compilers with mid-level optimizations

Certificate

TÜVRheinland CERTIFIED — Functional Safety — www.tuv.com ID 0000000000

No.: 968/K 220.01/17

Product tested	The Software (Firmware) SafeDEC for the ECU7 used for the emergency diesel generator	**Certificate holder**	MTU Friedrichshafen GmbH Maybachplatz 1 88045 Friedrichshafen Germany
Type designation	SafeDEC System for details see SafeDEC Revision Release List		
Codes and standards	IEC 60880:2006		IEC 61508-3:2010
Intended application	The software "SafeDEC" is designed to be embedded into the MPC565 microcontroller as software programmable processing unit of the existing MTU ECU7 hardware. The software qualification of the SafeDEC Software has shown that the requirements of IEC 60880 category A and the requirements of IEC 61508-3 Safety Capability 3 are fullfilled. The software can be used in applications up to SIL3, if the system and hardware requirements are fulfilled.		
Specific requirements	The Safety System Dataset shall be considered.		

Valid until 2022-11-10

The issue of this certificate is based upon an examination, whose results are documented in Report No. 968/K 220.01/17 dated 2017-11-10.
This certificate is valid only for products which are identical with the product tested.

TÜV Rheinland Industrie Service GmbH
Bereich Automation
Funktionale Sicherheit
Am Grauen Stein, 51105 Köln
Certification Body Safety & Security for Automation & Grid

Köln, 2017-11-10

Dr.-Ing. Thorsten Gantevoort

Memory Safety for C

- ✓ Violation of resource bounds
  - stack overflow

  → StackAnalyzer
  Abstract Interpretation

- ✓ Compliance to MISRA C
  - No missing alarms on semantical rules

  → Astrée
  Abstract Interpretation

- ✓ Data corruption in sequential execution
  - invalid pointer access and manipulation:
    buffer overflows, null pointer accesses, dangling pointers,
    out-of-bounds array accesses, ...

  → Astrée
  Abstract Interpretation

- ✓ Data corruption in concurrent execution
  - data races
  - inconsistent locking
  - access to variables reserved for other processes

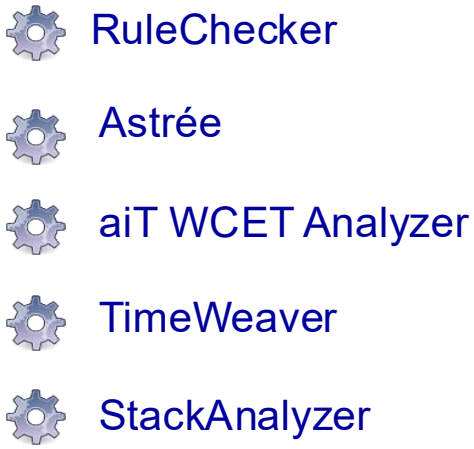
  → Astrée
  Abstract Interpretation

- ✓ Miscompilation
  - compiler silently generates incorrect code

  → CompCert
  formally verified
  optimizing compilation

Conclusion

- In safety-critical systems the absence of safety and security hazards has to be demonstrated.
- Static analysis crucial for safety and security
  - Recommended by all safety & security norms
  - ✓ Checking compliance to coding guidelines
  - ✓ Computing code metrics
  - ✓ Resource analysis: worst-case stack usage, worst-cast execution time
  - ✓ Runtime error analysis by sound static analysis
    - Absence of critical code defects can be proven:
      div/0, buffer overflow, null/dangling pointers, data races, deadlocks, …
    - Data and control flow coupling / Interference analysis
    - Fault avoidance



- CompCert provides unprecedented level of confidence in compiler correctness

email: kaestner@absint.com
http://www.absint.com