Verification Objectives for Cybersecurity and Safety – Friend or Foe?

Daniel Kästner
AbsInt GmbH, 2022
kaestner@absint.com

IARIA

AbsInt

Daniel Kästner

- 1993-1997: Study of Computer Science and Business Economics at Saarland University, Saarbrücken, Germany
- 1997: VDI Saar Master's Thesis Award
- 1997-2000: Graduate College at Saarland University
- 1998: Co-Founder of AbsInt GmbH
- 2000: PhD on Code Optimization for Embedded Processors
- 2000-2003: Research Associate at Saarland University & Senior Software Engineer at AbsInt
- 2001: SaarLB Science Award
- Since 2003: CTO of AbsInt
- Since 2014: Member of IEC-61508 Formal Methods Project Team
- Since 2017: Member of MISRA C Working Group
- Since 2020: Member of MISRA SQM Working Group
- Best paper awards: EDCC 2015, ERTS 2016, CYBER 2017, ERTS 2020, ERTS 2022

Daniel Kästner

- Research topics:
  - embedded systems
  - functional safety
  - cybersecurity
  - static program analysis
  - compiler technology
  - coding guidelines
  - software quality
  - formal methods
- Selected publications (2019-2022)
  - D. Kästner, L. Mauborgne, S. Wilhelm, C. Mallon, C. Ferdinand. Static Data and Control Coupling Analysis. In ERTS 2022: Embedded Real Time Software and Systems, 11th European Congress, Jun 2022, Toulouse, France. Available at the HAL open archive, URL: https://hal.archives-ouvertes.fr/hal-03694546
  - D. Kästner, M. Pister, C. Ferdinand. Obtaining DO-178C Certification Credits by Static Program Analysis (Best Paper Award). In ERTS 2022: Embedded Real Time Software and Systems, 11th European Congress, Jun 2022, Toulouse, France. Available at the HAL open archive, URL: https://hal.archives-ouvertes.fr/hal-03694553
  - R. Wilhelm, M. Pister, G. Gebhard, and D. Kästner. Testing Implementation Soundness of a WCET Analysis Tool. In Jian-Jia Chen, Ed., A Journey of Embedded and Cyber-Physical Systems, Springer Open Access, 2021. ISBN 978-3-030-47487-4 (eBook).
  - D. Kästner, L. Mauborgne, S. Wilhelm, C. Ferdinand. High-Precision Sound Analysis to Find Safety and Cybersecurity Defects (Best Paper Award). In ERTS 2020: Embedded Real Time Software and Systems, 10th European Congress, Jan 2020, Toulouse, France.
  - D. Kästner, L. Mauborgne, C. Ferdinand, H. Theiling. Detecting Spectre Vulnerabilities by Sound Static Analysis. In CYBER 2019: Proceedings of the Fourth International Conference on Cyber-Technologies and Cyber-Systems, Porto, 2019.
  - D. Kästner, B. Schmidt, M. Schlund, L. Mauborgne et al. Analyze this! Sound static analysis for integration verification of large-scale automotive software. SAE Technical Paper 2019-01-1246, 2019.
  - D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer et al. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In ERTS 2018 — Embedded Real Time Software and Systems, January 2018, Toulouse, France. <hal-01643290>

Embedded System Trends – Safety and Security

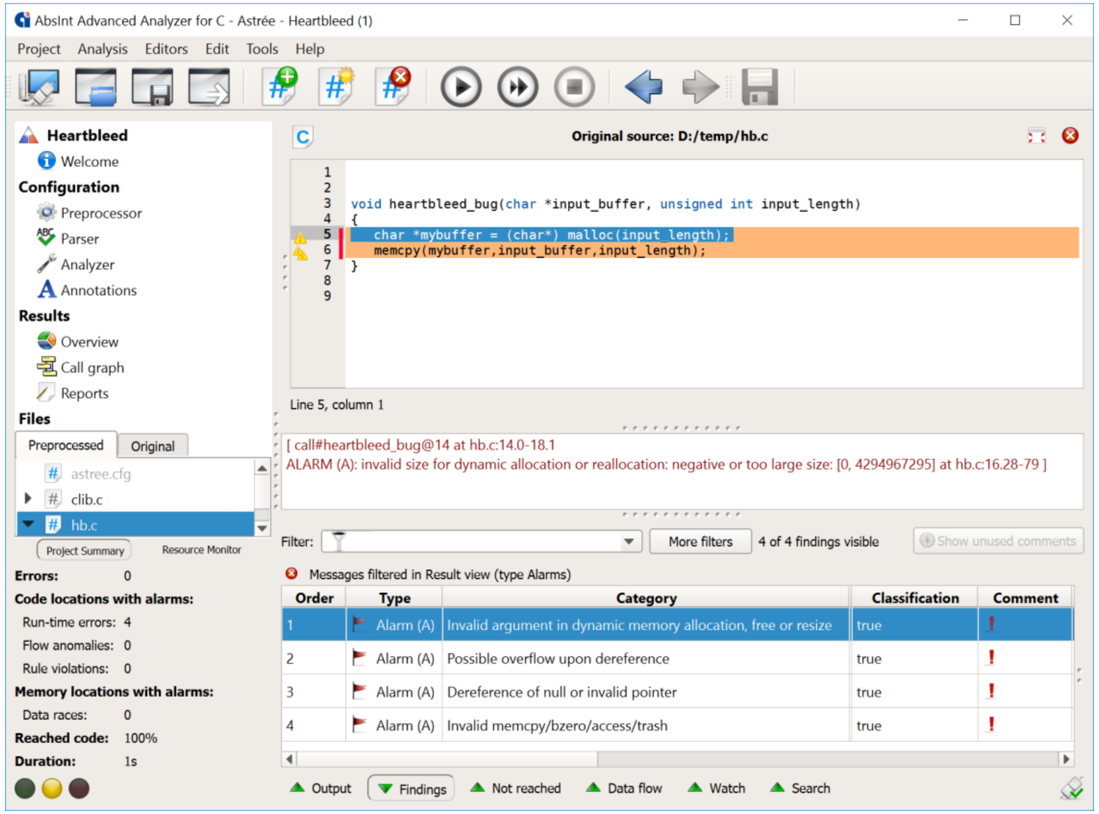
- Snowballing software complexity (2016: >100 million LOC per car [1])
- More and more safety-critical functionality in software
  - Autonomy: Highly automatic driving, Unmanned Aerial Vehicles, robotics, smart medical devices, …
- Increasing connectivity in safety-critical systems
  - Cloud-based services
  - 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)

⇨ Increasing risk of critical software defects

[1] Ondrej Burkacky, Johannes Deichmann, Georg Doll, Christian Knochenhauer. Rethinking car software and electronics architecture. Report McKinsey & Company, Feb. 2018.

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);
}
```



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

- 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

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
- Safety of the Intended Functionality – SOTIF
  - Absence of unreasonable risk due to hazards resulting from functional insufficiencies of the intended functionality

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

+ Security-relevant ISO 21434, …

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

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

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

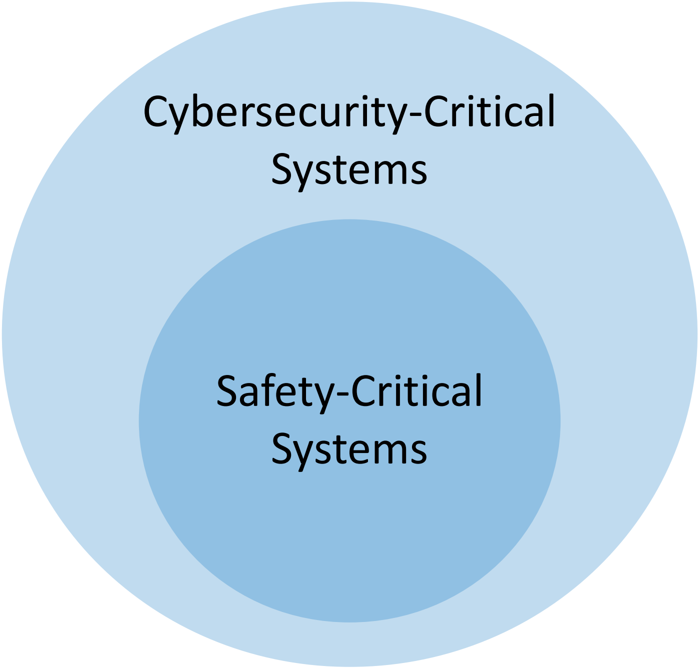
(Information-/Cyber-) Security Aspects

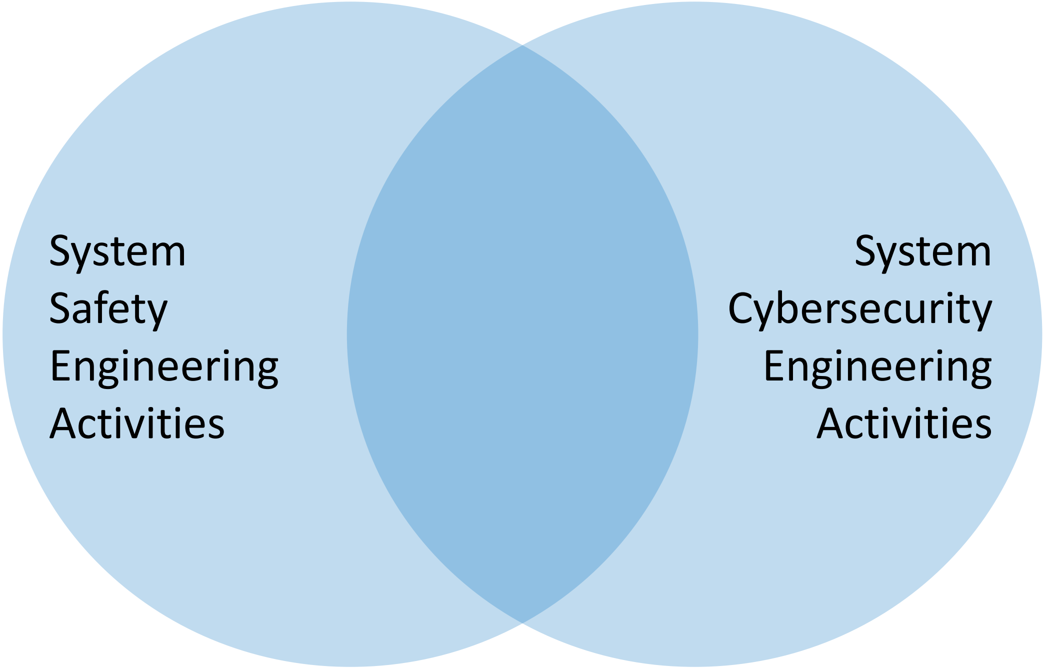
- Confidentiality
  - Information shall not be disclosed to unauthorized entities
  - ⇨ safety-relevant
- Integrity
  - Data shall not be modified in an unauthorized or undetected way
  - ⇨ safety-relevant
- Availability
  - Data is accessible and usable upon demand
  - ⇨ safety-relevant

\+ Safety

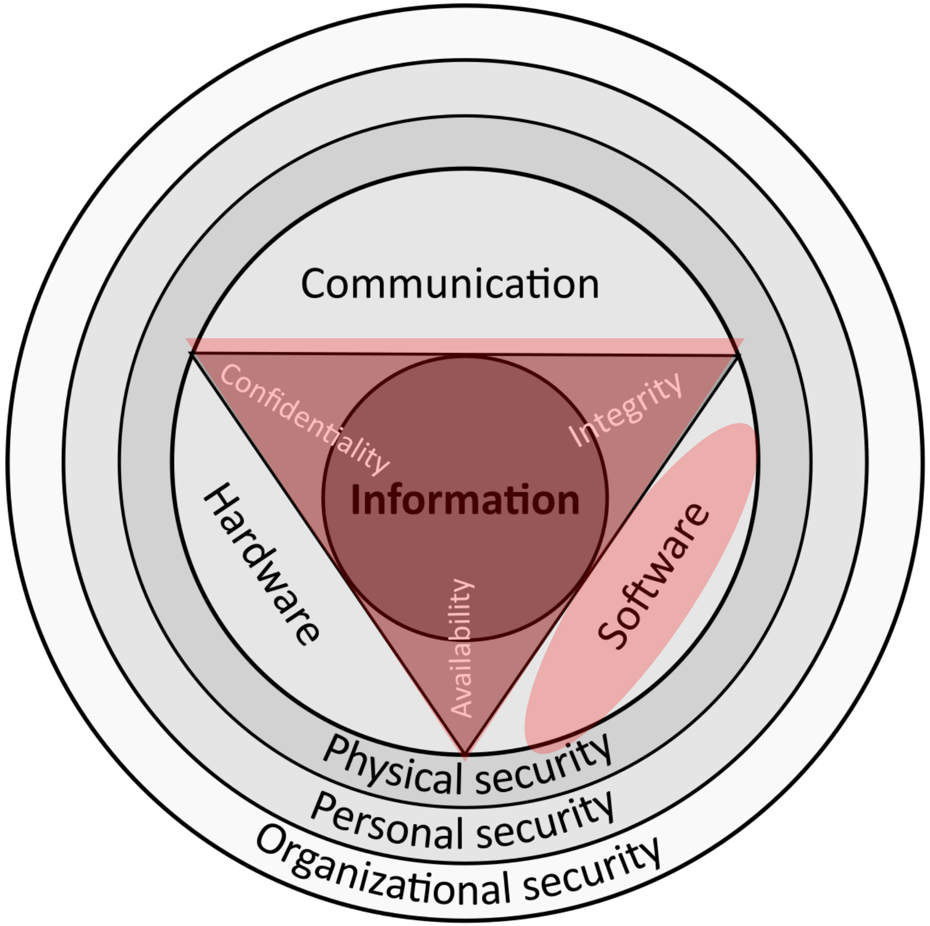
In some cases: not safe ⇒ not secure
In some cases: not secure ⇒ not safe

Relation between Safety and Cybersecurity





Scope of this Talk



Source: https://en.wikipedia.org/wiki/Information_security

Cybersecurity at the Source Code Level

- 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 shown with sound static analysis!

- Beyond runtime errors:
  - Coding guidelines
  - Data and Control Flow analysis
  - Taint analysis (data safety, impact analysis, …)
  - Side channel attacks
    - Spectre
  - …

- Security is more complex!
  - Safety: property of single execution traces
  - Security: property of sets of execution traces (hyperproperties)

SSP – Normative Landscape

Functional Safety: DO-178B/C, ISO 26262, IEC 61508, EN 50128, EN 62304, …

Security: SAE-J3061, ISO/SAE 21434, IEC TR 63069, IEC 62443,
ISO 15408, MDCG 2019-16, …

Performance / System Safety: ISO PAS 21448 DIS (SOTIF)

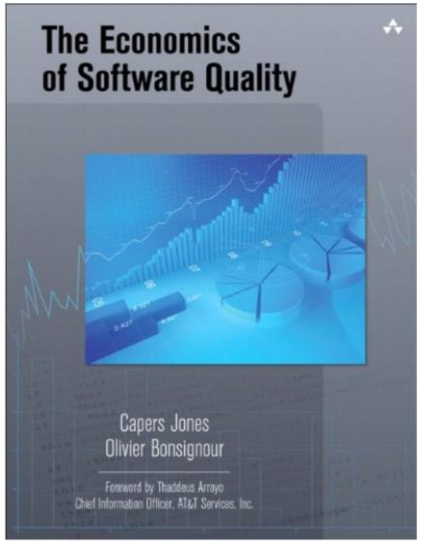
UL4600 (Autonomous products)

ISO NWIP TS5083 (Automated road driving systems)

(Product Safety: IATF 16949, Legislation (EU General Product Safety Directive,
FMVSS, Type Approval))

Bugs Happen

Size of SW projects in FP	Average Defect Potential	Defect Removal Efficiency
100	3,00	98%
10.000	6,25	93%
1.000.000	8,25	86%



1 FP (Function point) ~ 160 LOC (C language)
64 LOC (Ada)
32 LOC (C++)

Total potential defects correlated with defect removal efficiency for:
10.000 FP ~ 1.6 MLOC C-Code:

⇨ 62.500 defects

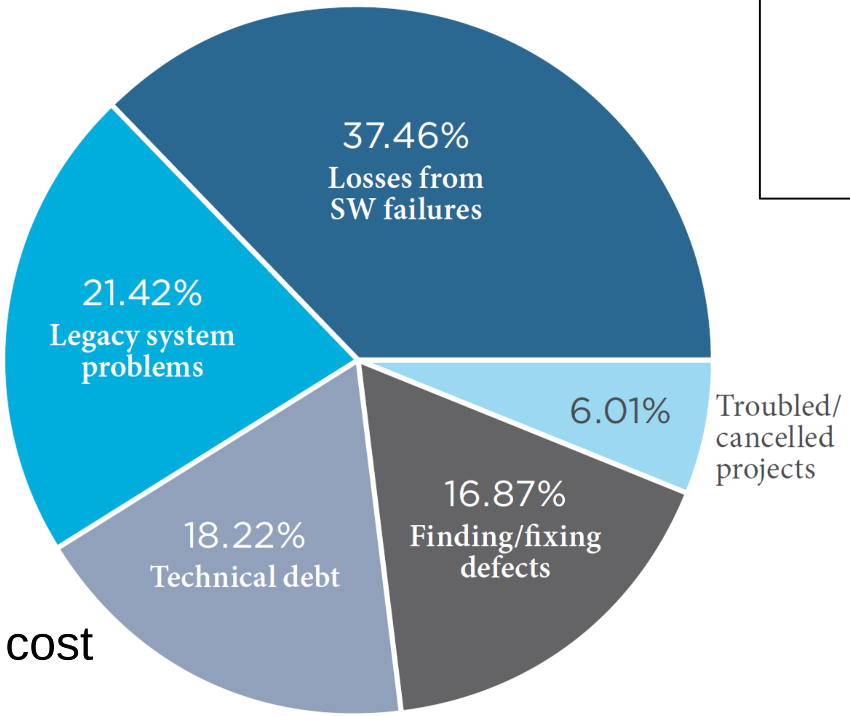
⇨ ~4.375 defects delivered to customer

Tables Tab. 6.9 / Tab. 6.10 / Tab. 6.11 / Tab. 6.12 (Numbers for Systems & Embedded Software Projects) from:

[4] Capers Jones, Olivier Bonsignour. The Economics of Software Quality. Addison-Wesley Professional; 2011.

Cost of Poor Software Quality

- Cost of poor quality software in US in 2018: ~2.84 trillion USD
- „The key strategy for reducing the cost of poor software quality is to find and fix problems and deficiencies as close to the source as possible, or better yet, prevent them from happening in the first place.“



Areas of cost

[1] Herb Krasner. The Cost of Poor-Quality Software in the US, CISQ, 2018.

Costs of Software Defects

- Speculative table on costs of software defects, considering, e.g.,
  - Toyota brake problem
  - NASA Mariner 1 failure
  - NASA Polar Lander failure
  - Therac-25 radiation poisoning
  - Ariane-5 explosion
  - Patriot missile targeting error
  - Chinook helicopter engine failure
  - F-22 Raptor flight control errors
  - Shutdown of Yorktown shipboard software
- Some software defects can be very expensive

Table 6.20. Approximate U.S. Annual Costs for Software Defects to Clients

Cost per Incident	MIS/Web Software	Systems & Embedded Software	Total Annual Incidents	Annual Cost of Incidents
> $1,000,000,000	5	2	7	$7,000,000,000
> $100,000,000	50	15	65	$6,500,000,000
> $10,000,000	100	50	150	$1,500,000,000
> $1,000,000	200	75	275	$275,000,000
> $100,000	1,000	250	1,250	$125,000,000
> $10,000	2,000	1,000	3,000	$30,000,000
> $1,000	6,000	5,000	11,000	$16,000,000
> $100	20,000	10,000	30,000	$3,000,000
TOTAL	29,355	16,392	45,747	$15,449,000,000

[4] Capers Jones, Olivier Bonsignour. The Economics of Software Quality. Addison-Wesley Professional; 2011.

Effectiveness of Software Defect Prevention Methods

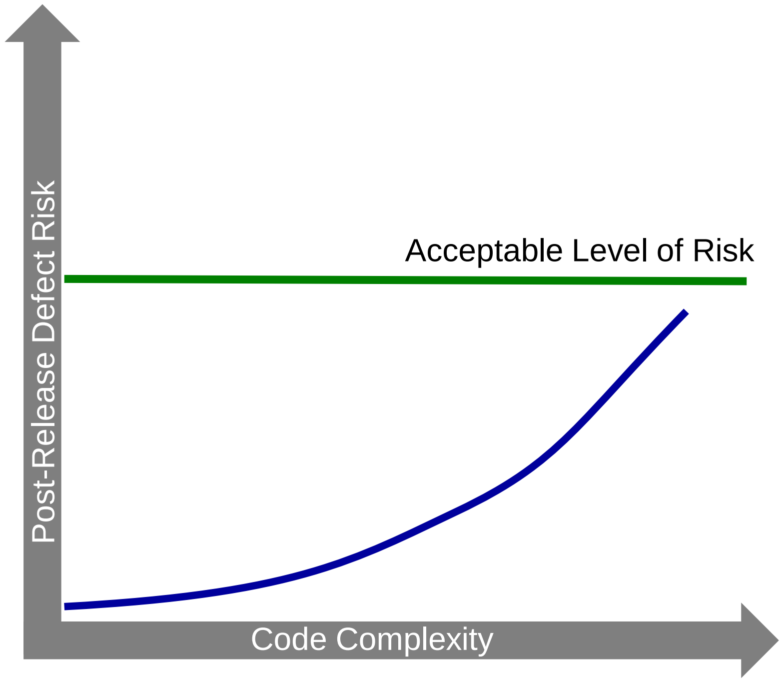
- Total of 65 methods estimated with static analysis ranked 7

	(Reductions in defects per function point for 1,000 function points)			
	Defect Prevention Methods (In Order of Effectiveness)	Defect Prevention Efficiency	Defects Potentials without Prevention	Defect Potentials with Prevention
1	Reuse (certified sources)	85.00%	5.00	0.75
2	Inspections (formal)	60.00%	5.00	2.00
3	Quality Function Deployment (QFD)	57.50%	5.00	2.13
4	Prototyping—functional	52.00%	5.00	2.40
5	Risk analysis (automated)	48.00%	5.00	2.60
6	PSP/TSP	44.00%	5.00	2.80
7	Static analysis of source code	44.00%	5.00	2.80
8	Root cause analysis	41.00%	5.00	2.95
9	Quality in all status reports	40.00%	5.00	3.00
10	Joint Application Design (JAD)	40.00%	5.00	3.00
11	Test-driven development	37.00%	5.00	3.15
12	CMMI 5	37.00%	5.00	3.15

[4] Capers Jones, Olivier Bonsignour. The Economics of Software Quality. Addison-Wesley Professional; 2011.

Software Complexity and Defect Risk

- Defect potential increases with complexity of SW
- Defect removal efficiency decreases with complexity of SW
- Required hardware complexity increases with complexity of SW
  - Non-deterministic interference effects on high-end multicore architectures



⇨ Requires appropriate development processes and usage of highly efficient and powerful software tools

Capers Jones, Olivier Bonsignour. The Economics of Software Quality. Addison-Wesley Professional; 2011.

ISO 26262 – Modelling and Coding Guidelines

Table 1 — Topics to be covered by modelling and coding guidelines

	Topics	ASIL A	ASIL B	ASIL C	ASIL D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques[d]	+	+	++	++
1e	Use of well-trusted design principles[e]	+	+	++	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++
1i	Concurrency aspects[f]	+	+	+	+

[a] An appropriate compromise of this topic with other requirements of this document may be required.

[b] The objectives of topic 1b include:

— Exclusion of ambiguously-defined language constructs which may be interpreted differently by different modellers, programmers, code generators or compilers.

— Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.

— Exclusion of language constructs which could result in unhandled run-time errors.

[c] The objective of topic 1c is to impose principles of strong typing where these are not inherent in the language.

[f] Concurrency of processes or tasks is not limited to executing software in a multi-core or multi-processor runtime environment.

Excerpt from: Sec. 5.4.3 – General topics for the product development at the software level,
ISO 26262-6 Road vehicles - Functional safety – Part 6: Product development: Software Level, 2018.

ISO 26262 – SW Unit Design and Implementation

Table 6 — Design principles for software unit design and implementation

Principle		ASIL A	ASIL B	ASIL C	ASIL D
1a	One entry and one exit point in sub-programmes and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	++	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Restricted use of pointers[a]	+	++	++	++
1g	No implicit type conversions[a]	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps[a]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Principles 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

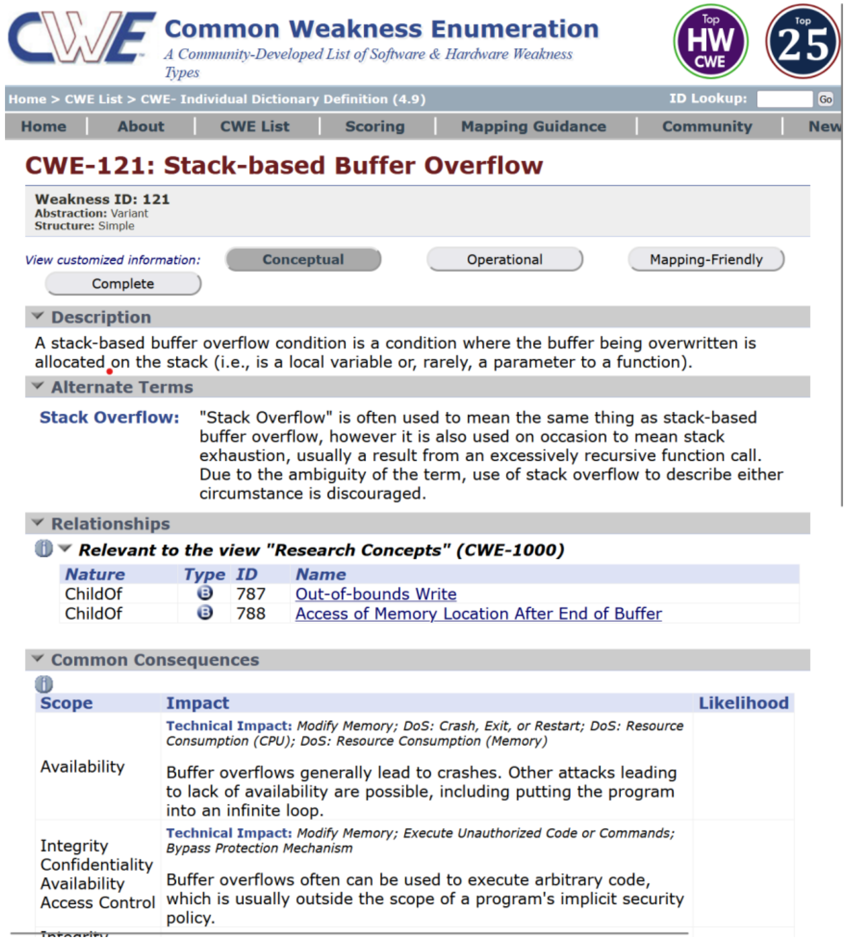
NOTE For the C language, MISRA C[3] covers many of the principles listed in Table 8.

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

Sources of Security Vulnerabilities in C

- Many security vulnerabilities due to undefined / unspecified behaviors in the programming language semantics:
  - Stack-based buffer overflows
  - Heap-based buffer overflows
  - Invalid pointer accesses (null, dangling, …)
  - Uninitialized memory accesses
  - Integer errors
  - Format string vulnerabilities
  - Concurrency defects (TOCTOU races, …)

⇨ Consequences: denial-of-service, code injection, data breach



ISO 21434 – Road vehicles — Cybersecurity Engineering

[RQ-10-04] If design, modelling or programming notations or languages are used for the cybersecurity specifications or their implementation, the following shall be considered when selecting such a notation or language:

a) an unambiguous and comprehensible definition in both syntax and semantics;

b) support for achievement of modularity, abstraction and encapsulation;

c) support for the use of structured constructs;

d) support for the use of secure design and implementation techniques;

e) ability to integrate already existing components; and

EXAMPLE 3 Library, framework, software component written in another language.

f) resilience of the language against vulnerabilities due to its improper use.

EXAMPLE 4 Resilience against buffer overflows.

NOTE 6 For software development, implementation includes coding using programming languages.

ISO 21434 – Road vehicles — Cybersecurity Engineering

[RQ-10-05] Criteria (see [RQ-10-04]) for suitable design, modelling or programming languages for cybersecurity that are not addressed by the language itself shall be covered by design, modelling and coding guidelines, or by the development environment.

EXAMPLE 5 Use of MISRA C:2012 [17] or CERT C [18] for secure coding in the "C" programming language.

EXAMPLE 6 Criteria for suitable design, modelling and programming languages:

— use of language subsets;

— enforcement of strong typing; and/or

— use of defensive implementation techniques.

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 – Methods for 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.

ISO 262626 – Methods for Software Integration Verification

Table 10 — Methods for verification of software integration

	Methods	ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage evaluation[c, d]	++	++	++	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++
1f	Verification of the control flow and data flow	+	+	++	++
1g	Static code analysis[f]	++	++	++	++
1h	Static analyses based on abstract interpretation[g]	+	+	+	+

[a] The software requirements allocated to the architectural elements are the basis for this requirements-based test.

[b] In the context of software integration testing, fault injection test means to introduce faults into the software for the purposes described in 10.4.3 and in particular to test the correctness of hardware-software interface related to safety mechanisms. This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software interfaces). Fault injection can also be used to verify freedom from interference.

[c] To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for programme and data) and the bandwidth of communication links (e.g. data buses) have to be determined.

[d] Some aspects of the resource usage evaluation can only be performed properly when the software integration tests are executed on the target environment or if the emulator for the target processor adequately supports resource usage tests.

[e] This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

[f] Static analyses are a collective term which includes analysis such as architectural analyses, analyses of resource consumption and searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines, if not already verified at the unit level.

[g] Static analyses based on abstract interpretation are a collective term for extended static analysis which also 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, if not already verified at the unit level.

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

ISO 21434 – Road vehicles — Cybersecurity Engineering

[RQ-10-10] The integration and verification activities of [RQ-10-09] shall be specified considering:

a) the defined cybersecurity specifications;

b) configurations intended for series production, if applicable;

c) sufficient capability to support the functionality specified in the defined cybersecurity specifications; and

d) conformity with the modelling, design and coding guidelines of [RQ-10-05], if applicable.

NOTE 1 This can include the vehicle integration and verification.

NOTE 2 Methods for verification can include:

— requirements-based test;

— interface test;

— resource usage evaluation;

— verification of the control flow and data flow;

— dynamic analysis; and/or

— static analysis.

Example Safety/Security Goal Conflicts

- Car locking
  - Safety: Unlock car in case of accident.
  - Security: Lock car when engine not running for some time.

- Update policy
  - Safety: Updates only available after full integration verification
  - Security: Provide quick patches to react to cybersecurity threads

- Connectivity policy
  - Safety: Enforce minimal connectivity
  - Security: Enable over-the-air updates for quick reactivity

Aligning Safety and Security Processes

- Safety and cybersecurity processes have to be aligned.
- Example for concept phase:

- HARA (Hazard and Risk Analysis)
  - Potential item hazards
  - Potential vehicle level hazards
  - Potential worst-case hazard scenario
  - ASIL determination
    - Severity
    - Exposure
    - Controllability
  - Safety Goals

Touching points

- TARA (Thread and Risk Analysis)
  - Potential item threats
  - Potential vehicle level threats
  - Potential worst-case thread scenario
  - CAL determination
    - Severity
    - Attack likelihood
    - Controllability
  - Cybersecurity Goals

Coding Guidelines

- "Safety"
  - MISRA C:2004, MISRA C:2012, MISRA C++:2008
    - Guidelines to define a language subset to avoid or reduce the risk for programming errors
- "Security"
  - ISO/IEC TS 17961:2013 "C Secure"
    - MISRA C:2012 Addendum 2 gives mapping to C Secure
  - SEI CERT C Coding Standard / CERT C++
    https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
    - Rules to facilitate developing safe, reliable, and secure systems
    - MISRA C:2012 Addendum 3 gives mapping to CERT C
  - MITRE Common Weakness Enumeration CWE
    https://cwe.mitre.org

Coding Guidelines – Safety vs. Security

- MISRA C:2012 vs. ISO/IEC TS 17961:2013
  - Only 4 "C Secure" rules not addressed by MISRA C:2012
  - Those have been added with MISRA C:2012 Amendment 1
- SEI CERT C – Example Rules:
  - EXP.33 Do not read uninitialized memory
  - EXP.34 Do not dereference null pointers
  - INT.32 Ensure that operations on signed integers do not result in overflow
  - INT.33 Ensure that division and reminder operations do not result in divide-by-zero errors
  - ➢ Run-time errors due to undefined/unspecified behaviors
  - ⇨ Strong overlap with safety-oriented rule sets (cf. MISRA C Addendum 3)
- Common Weakness Enumeration CWE
  - Similar
- Adaptive Autosar C++14 Coding Guidelines, …

MISRA C:2012 Guideline Classification

- Directives (17)
  - No fully automatic compliance check on source code
    - Criteria not precisely defined
    - Additional activities needed (checking external documentation, …)
  - Example (Dir 3.1.): All code shall be traceable to documented requirements
- Rules (156)
  - Automatic compliance check possible
  - Scope: system (49) vs. single-translation-unit (107)
  - Algorithmic class: undecidable vs. decidable

MISRA C Guideline Classification

- Decidable rules (119)
  - syntactical property
  - Example – Rule 2.7.
    There should be no unused parameters in functions.

- Undecidable rules (37)
  - semantical property $\Rightarrow$ absence of defect
  - Example – Rule 9.1:
    The value of an object with automatic storage duration shall not be read before it has been set

Entscheidungsproblem & Halting Problem

- Decision problem: find an algorithm to determine for every possible parameter instance whether a certain parametric statement is true or false in a given axiomatic system [Hilbert 1928].
- Halting Problem: Given some algorithm and some input to the algorithm (both together are the parameter) does the algorithm halt when run on the given input?
- The Halting Problem is undecidable [Turing 1937]

⇨ A Turing Machine that decides whether some (other) Turing Machine halts when run on an input cannot exist.

- Rice Theorem: All non-trivial semantic statements about Turing machines (or programs) are undecidable.

The Halting Problem

- C is a Turing-complete programming language, i.e., any possible Turing machine can be implemented as a C program.
- Does this program terminate?

```
int Collatz(int c)
{
  int n=c;
  while (1 != n) {
    if (0 == n%2) n = n/2;
    else n = (3*n)+1;
  }
  return 0;
}
```

Undecidables Rules in MISRA C:2012 – Examples

- Dir 4.7 (required, undecidable, system)
  If a function returns error information, then that error information shall be tested.
- Rule 2.1 (required, undecidable, system)
  A project shall not contain unreachable code.
- Rule 9.1 (mandatory, undecidable, system)
  The value of an object with automatic storage duration shall not be read before it has been set.
- Rule 14.3 (required, undecidable, system)
  Controlling expressions shall not be invariant.
- Rule 17.2 (required, undecidable, system)
  No recursive function calls

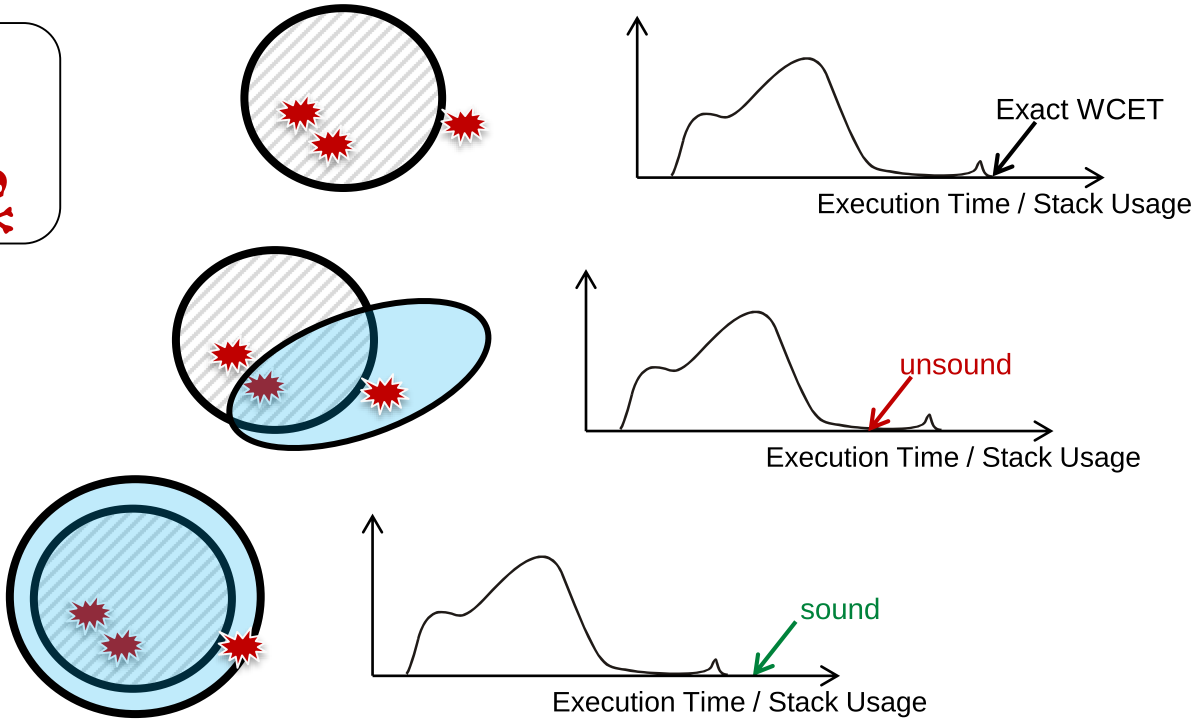
Static Program Analysis

- 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

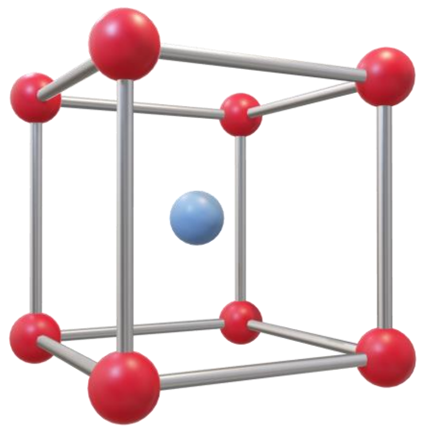
Question: Is there an error in the program?
- False positive: answer wrongly "Yes"
- False negative: answer wrongly "No"

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

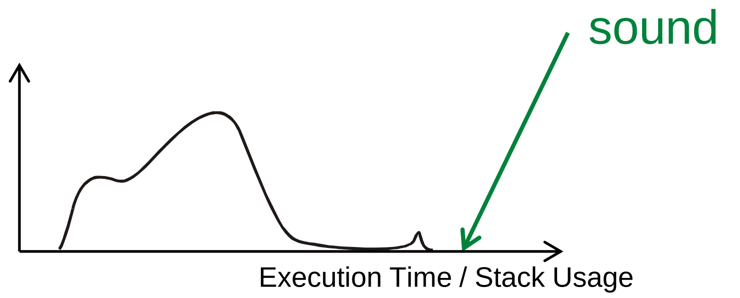
☑Sound / Abstract Interpretation-based
  - False positives: possible
  - No false negatives ⇨ Soundness
    No defect missed

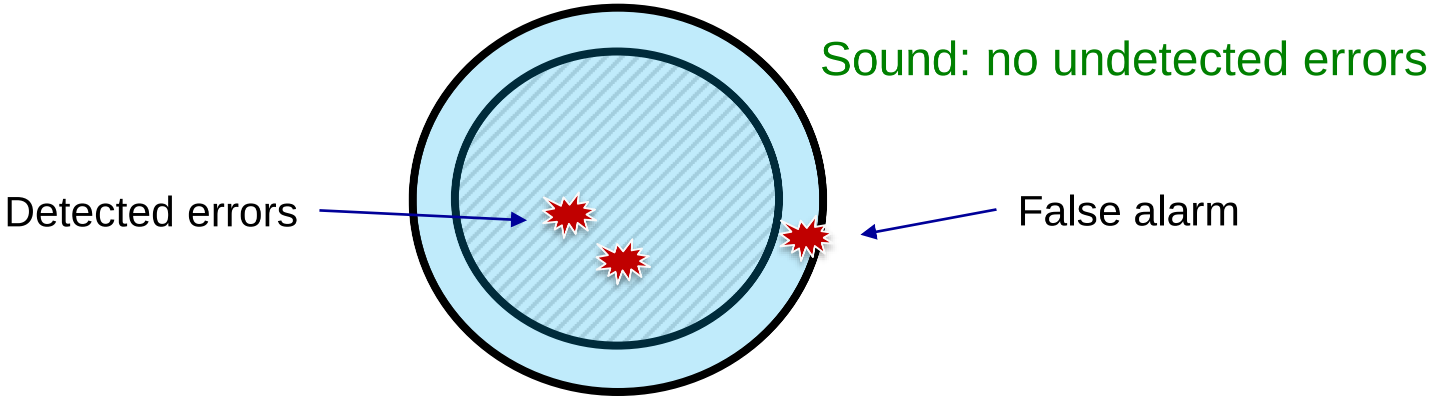


Abstract Interpretation



- Semantics based methodology for program analysis
- Formal method – supports correctness proofs
  - Efficiency: scales to real-life industry applications due to abstractions
  - Soundness:
    - Correctness of abstractions proven.
    - Never fail to report a defect from the class of defects under analysis
  - Safety: over-approximate the program semantics. Some precision may be lost, but always on the safe side.



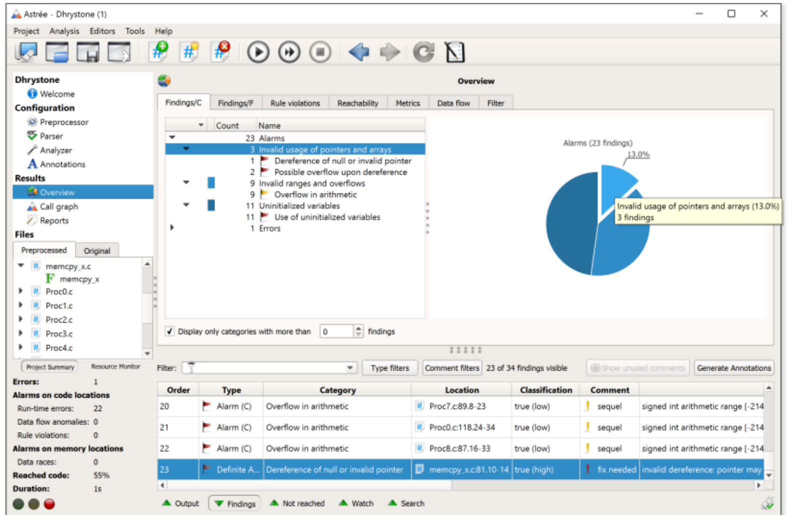


Analysis Depth

- 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:
    - Definite alarm on **`a/b`** (b==0): division by zero will happen in given context
    - 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

Runtime Error Analysis

- 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, …
  - Data and control flow analysis (data and control coupling), interference analysis, alias analysis
  - Taint analysis (data safety / security), SPECTRE detection
  - + User-defined assertions, unreachable code, non-terminating loops
  - + Check coding guidelines – RuleChecker included: MISRA C/C++, Adaptive AUTOSAR C++, CERT C/C++, CWE, ISO TS 17961 (standalone operation: QA-MISRA)
  - + Automatic support for ARINC653/OSEK/AUTOSAR OS configurations
  - ➢ Supports C and safety-critical C++

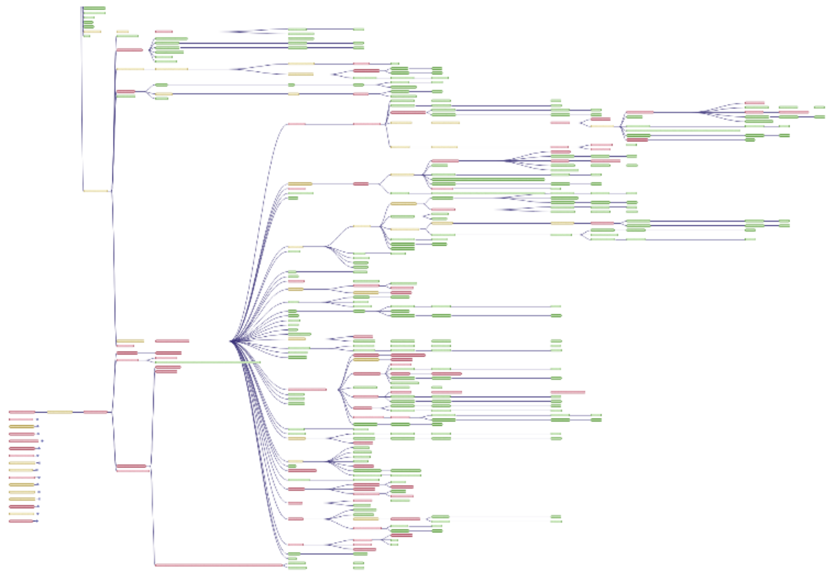


* Defects due to undefined / unspecified behaviors of the programming language

Runtime Error Analysis
Data & Control Flow Analysis

Data and Control Flow Analysis

- 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
  - Defined at software component level
- Soundness: no data/control flow is missed
  - Aware of data and function pointers, task interference, …



Data and Control Coupling Analysis

- Purpose: determine **effective** data and control flow between **software components**
  - May be desired or undesired, to be further investigated
- Behaviors undefined or unspecified in the programming language may have undefined / unspecified effects on data and control flow, hence, have to be considered as control / data flow defect.
- Example: Division by 0, causing a trap, leading to program termination.

⇨ Sound runtime error analysis is prerequisite for data and control flow/coupling analysis

Static Taint Analysis

- Purpose: Static analysis to track flow of tainted values through program.
- Concepts:
  - Taint source: origin of tainted values
  - Taint sink: memory location: operands and arguments to be protected from tainted values
  - Sanitization: remove taint from value, e.g. by replacement or termination
- User interaction to identify tainted sources and sinks.
- Typical applications:
  - Information Flow (Confidentiality / Information Leaks)
  - Propagation of Error Values (Data and Control Flow)
- Astrée:
  - Universal user-configurable taint analysis
  - Detection of Spectre V1/V1.1/SplitSpectre vulnerabilities
  - Data and Control Coupling / Interference Analysis

Component Tainting

- Taint analysis allows to track the flow of values.
- Computing data dependences:
  - Taint all variables (global, static and local) of component C with hue^{C}
  - All variables of a component C are taint sinks for hues hue^{X} of all components $X \neq C$.
  - All variable reads in a component C are interpreted as taint sink for $hue^{X}, X \neq C$.

  ⇨ Automatic notification of out-of-component accesses to values from C.

  ⇨ Supports sanitization: values from C legal to access via gateway function f.
- Computing control dependences:
  - Taint sinks can only be
    - Guards, e.g., in conditional statements, loops or switch statements, and in
    - Function pointer dereferences

SPECTRE

- Side channel attack:
  Speculative execution
  (primarily branch prediction
  on array bound accesses)
  exploited to load confidential
  data in the cache from where
  they are leaked.
- Billions of processors affected:
  ARM, Intel, AMD, IBM, …


The Washington Post
Democracy Dies in Darkness
TECHNOLOGY
Huge security flaws revealed — and tech companies can barely keep up
By Craig Timberg, Elizabeth Dwoskin and Hamza Shaban
January 5, 2018 at 9:39 p.m. EST

The New York Times
Researchers Discover Two Major Flaws in the World's Computers

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY
Alerts and Tips   Resources
National Cyber Awareness System > Alerts > Meltdown and Spectre Side-Channel Vulnerability Guidance
Alert (TA18-004A)
Meltdown and Spectre Side-Channel Vulnerability Guidance

- Many variants:
  Spectre Variant V1, V1.1, SplitSpectre, V2, V4, ret2spec, Spectre-RSB, more still being discovered
- As of today: no protection known without CPU architecture changes

Spectre Classes

- Transient execution attacks: transfer microarchitectural state changes caused by the execution of transient instructions (i.e., whose result is never committed to architectural state) to an observable architectural state.
  - Meltdown: transient out-of-order instructions after CPU exception
  - Spectre: exploit branch misprediction events

- Spectre types
  - Spectre-PHT: Pattern History Table ▷ Spectre V1, V1.1, SplitSpectre
  - Spectre-BTB: Brant Target Buffer ▷ Spectre V2
  - Spectre-STL: Store-to-Load Forwarding ▷ Spectre V4
  - Spectre-RSB: Return Stack Buffer ▷ ret2spec, Spectre-RSB

Vulnerable Code and Fix

```
ErrCode vulnerable1 (unsigned idx )
{
  if (idx >= arr1.size) {
    return E_INVALID_PARAMETER;
  }
  unsigned u1 = arr1.data[idx];
  ...
  unsigned u2 = arr2.data[u1];
  ...
}
```

Untrusted data (attacker-controlled)

Can be executed with out-of-range values after mis-predicted branches

Value read from `arr1` is used to index `arr2`. The memory access modifies the cache.

Timing attack can identify cache cell with hit, which leaks `u1`, ie., the contents of `arr1`.

Fix

```
ErrCode vulnerable1 (unsigned idx)
{
  if (idx >= arr1.size) {
    return E_INVALID_PARAMETER;
  }
  unsigned fidx = FENCEIDX(idx,arr1.size);
  ...
  unsigned u1 = arr1.data[fidx];
  ...
  unsigned u2 = arr2.data[u1];
  ...
}
```

`FENCEIDX` maps `idx` into the feasible array range.

Taint Analysis for Spectre

- Two taints: **`controlled`** and **`dangerous`**
- Manual tainting of user-controlled values as **`controlled`**
  - E.g., all parameters of “public” API functions

- Automatic detection of comparison of **`controlled`** values with bounds

⇨ Taint automatically changed from **`controlled`** to **`dangerous`**

- Remove **`dangerous`** taint at end of speculative execution window.
  Architecture-independent solution:

⇨ Automatic reset to **`controlled`** at control flow join

Spectre V1/V1.1/SplitSpectre Detection

```
volatile int controlled;
__ASTREE_volatile_input((controlled; [1,2]));
int victim_function( size_t x) {
  if ( x < array1_size ) {
    temp &= array2 [array1[ x ] * 512];
  }
  return x;
}
void main(){
  unsigned int val, retval;
  init(&val);           //reads val from the environment
  __ASTREE_taint((val; controlled));
  retval = victim_function( val );
}
```

ALARM: Spectre vulnerability

- No complete protection but attack surface can be reduced

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

Conclusion

- More and more embedded applications are safety-critical and/or mission-critical
- Preventing safety and security hazards is essential to build trust
- Safety and security goals have to be aligned, often compatible
- Coding guidelines to minimize programming errors needed
- Sound static analysis crucial for safety and security
  - Defect prevention:
    no defect shipped to customer ⇨ no callback, no liability lawsuits
  - Absence of critical code defects can be proven
  - No runtime errors: "pretty good security"
- Inherent complexity of security is higher
  - Additional measures needed
  - Data and control flow analysis
  - Taint analysis, e.g., for detecting Spectre vulnerabilities
  - …

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