Verification Objectives for Cybersecurity and Safety – Friend or Foe?

Daniel Kästner
AbsInt GmbH, 2022
kaestner@absint.com
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
Daniel Kästner

- Research topics:
  - embedded systems
  - functional safety
  - cybersecurity
  - static program analysis
  - compiler technology
  - coding guidelines
  - software quality
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

The Internet of things (IoT) describes the network of physical objects—“things”—that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet.[1][2][3][4]

The definition of the Internet of things has evolved due to the convergence of multiple technologies, real-time analytics, machine learning, commodity sensors, and embedded systems.[1] Traditional fields of embedded systems, wireless sensor networks, control systems, automation (including home and building automation), and others all contribute to enabling the Internet of things. In the consumer market, IoT technology is most synonymous with products pertaining to the concept of the "smart home", including devices and appliances (such as lighting fixtures, thermostats, home security systems and cameras, and other home appliances) that support one or more common ecosystems, and can be controlled via devices associated with that ecosystem, such as smartphones and smart speakers.

There are a number of serious concerns about dangers in the growth of IoT, especially in the areas of privacy and security, and consequently industry and governmental moves to address these concerns have begun including the development of international standards.[5]
A Security Issue?

```c
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

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

---

**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, …

---

- 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

Cybersecurity-Critical Systems

Safety-Critical Systems

System Safety Engineering Activities

System Cybersecurity Engineering Activities
Scope of this Talk

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

- 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

<table>
<thead>
<tr>
<th>Size of SW projects in FP</th>
<th>Average Defect Potential</th>
<th>Defect Removal Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3,00</td>
<td>98%</td>
</tr>
<tr>
<td>10,000</td>
<td>6,25</td>
<td>93%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,25</td>
<td>86%</td>
</tr>
</tbody>
</table>

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:  
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

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

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

Effectiveness of Software Defect Prevention Methods

- Total of 65 methods estimated with static analysis ranked 7

<table>
<thead>
<tr>
<th>Defect Prevention Methods (In Order of Effectiveness)</th>
<th>Defect Prevention Efficiency</th>
<th>Defect Potentials without Prevention</th>
<th>Defect Potentials with Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reuse (certified sources)</td>
<td>85.00%</td>
<td>5.00</td>
<td>0.75</td>
</tr>
<tr>
<td>2. Inspections (formal)</td>
<td>60.00%</td>
<td>5.00</td>
<td>2.00</td>
</tr>
<tr>
<td>3. Quality Function Deployment (QFD)</td>
<td>57.50%</td>
<td>5.00</td>
<td>2.13</td>
</tr>
<tr>
<td>4. Prototyping—functional</td>
<td>52.00%</td>
<td>5.00</td>
<td>2.40</td>
</tr>
<tr>
<td>5. Risk analysis (automated)</td>
<td>48.00%</td>
<td>5.00</td>
<td>2.60</td>
</tr>
<tr>
<td>6. PSF/TSP</td>
<td>44.00%</td>
<td>5.00</td>
<td>2.80</td>
</tr>
<tr>
<td>7. Static analysis of source code</td>
<td>44.00%</td>
<td>5.00</td>
<td>2.80</td>
</tr>
<tr>
<td>8. Root cause analysis</td>
<td>41.00%</td>
<td>5.00</td>
<td>2.95</td>
</tr>
<tr>
<td>9. Quality in all status reports</td>
<td>40.00%</td>
<td>5.00</td>
<td>3.00</td>
</tr>
<tr>
<td>10. Joint Application Design (JAD)</td>
<td>40.00%</td>
<td>5.00</td>
<td>3.00</td>
</tr>
<tr>
<td>11. Test-driven development</td>
<td>37.00%</td>
<td>5.00</td>
<td>3.15</td>
</tr>
<tr>
<td>12. CMMI 5</td>
<td>37.00%</td>
<td>5.00</td>
<td>3.15</td>
</tr>
</tbody>
</table>

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 usage of highly efficient and powerful software tools
### Table 1 — Topics to be covered by modelling and coding guidelines

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Enforcement of low complexity</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b Use of language subsets</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1c Enforcement of strong typing</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1d Use of defensive implementation techniques</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1e Use of well-trusted design principles</td>
<td></td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1f Use of unambiguous graphical representation</td>
<td></td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1g Use of style guides</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1h Use of naming conventions</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1i Concurrency aspects</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

*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.*
### Table 6 — Design principles for software unit design and implementation

<table>
<thead>
<tr>
<th>Principle</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a One entry and one exit point in sub-programmes and functions&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++ ++ ++ ++</td>
</tr>
<tr>
<td>1b No dynamic objects or variables, or else online test during their creation&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1c Initialization of variables</td>
<td>++ ++ ++ ++</td>
</tr>
<tr>
<td>1d No multiple use of variable names&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++ ++ ++ ++</td>
</tr>
<tr>
<td>1e Avoid global variables or else justify their usage&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+ + ++ ++</td>
</tr>
<tr>
<td>1f Restricted use of pointers&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1g No implicit type conversions&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1h No hidden data flow or control flow</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1i No unconditional jumps&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++ ++ ++ ++</td>
</tr>
<tr>
<td>1j No recursions</td>
<td>+ + ++ ++</td>
</tr>
</tbody>
</table>

<sup>a</sup> 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<sup>31</sup> covers many of the principles listed in Table 8.
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
[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.
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.
ISO 26262 – Methods for Software Unit Verification

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Walk-through⁹</td>
<td></td>
<td>++</td>
<td></td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>1b Pair-programming⁹</td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>1c Inspection⁹</td>
<td></td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1d Semi-formal verification</td>
<td></td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1e Formal verification</td>
<td></td>
<td>0</td>
<td>o</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>1f Control flow analysis</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1g Data flow analysis</td>
<td></td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1h Static code analysis</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1i Static analyses based on abstract interpretation</td>
<td></td>
<td>++</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>1j Requirements-based test</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1k Interface test</td>
<td></td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1l Fault injection test</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>1m Resource usage evaluation</td>
<td></td>
<td>+</td>
<td>+</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>1n Back-to-back comparison test between model and code, if applicable</td>
<td></td>
<td>+</td>
<td>+</td>
<td>++</td>
<td></td>
</tr>
</tbody>
</table>

⁹ For model-based development these methods are applied at the model level, if evidence is available that justifies confidence in the code generator used.

Excerpt from:
Excerpt from:
[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
Safety/Security 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

- **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”
  - 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.
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?

```c
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**
    - **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

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

- **Exact WCET**
- **Execution Time / Stack Usage**
  - Sound
  - Unsound
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.
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
  + Automatic support for ARINC653/OSEK/AUTOSAR OS configurations
    - Supports C and safety-critical C++

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

- 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];
    ... }

\[\text{Fix}\]

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];
    ... }

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, i.e., the contents of arr1.

\[\text{FENCEIDX}\] 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

```c
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);
}
```

- No complete protection but attack surface can be reduced
## Efficiency

<table>
<thead>
<tr>
<th></th>
<th>Time to Analyze</th>
<th>Frequency of Analyses</th>
<th>Who runs Analyses</th>
<th>Level of Automation</th>
<th>False Alarm Rate</th>
<th>Defect Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>weeks</td>
<td>per release</td>
<td>QA</td>
<td>manual</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Sound Static Analysis</td>
<td>minutes</td>
<td>per change</td>
<td>everyone</td>
<td>automatic</td>
<td>low</td>
<td>complete</td>
</tr>
</tbody>
</table>

### Cybersecurity and Safety - Friend or Foe?
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
  - ...

Cybersecurity and Safety - Friend or Foe?