# An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection

Research Webinar @ BINSEC Team, Université Paris-Saclay

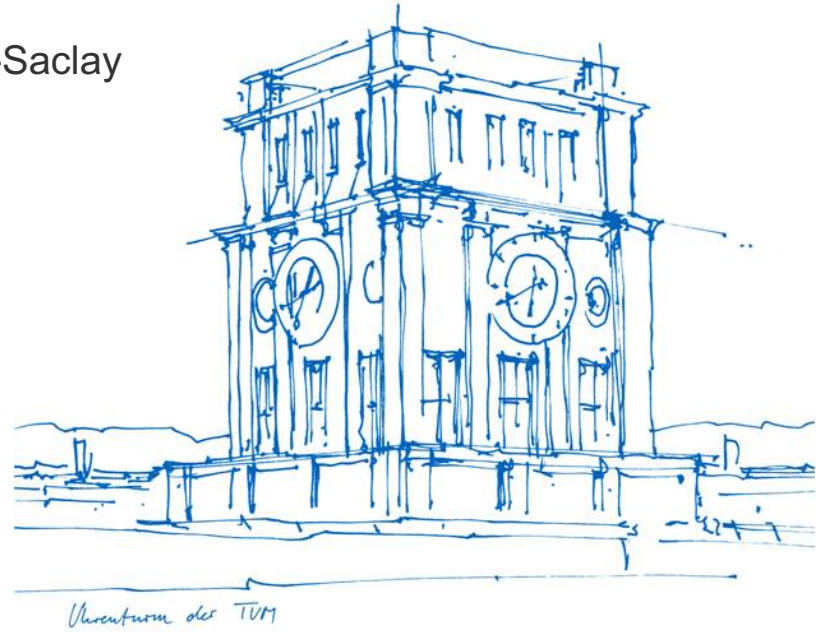
Seminar Talk @ Research Training Group ConVeY

Stephan Lipp

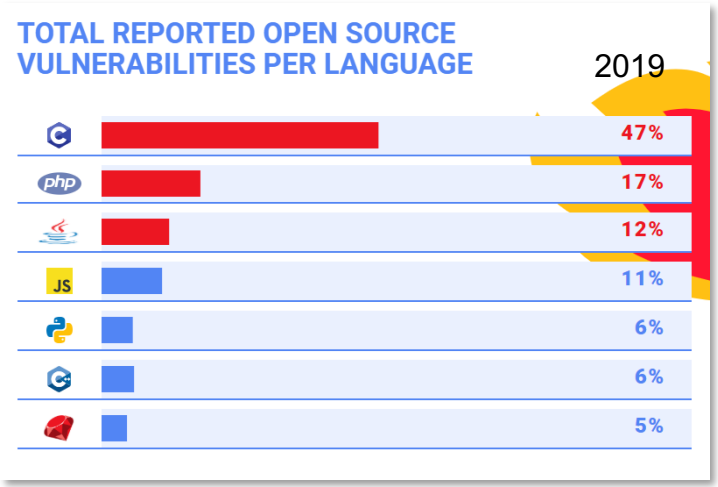
Technical University of Munich

Chair of Software & Systems Engineering

27/01/2023

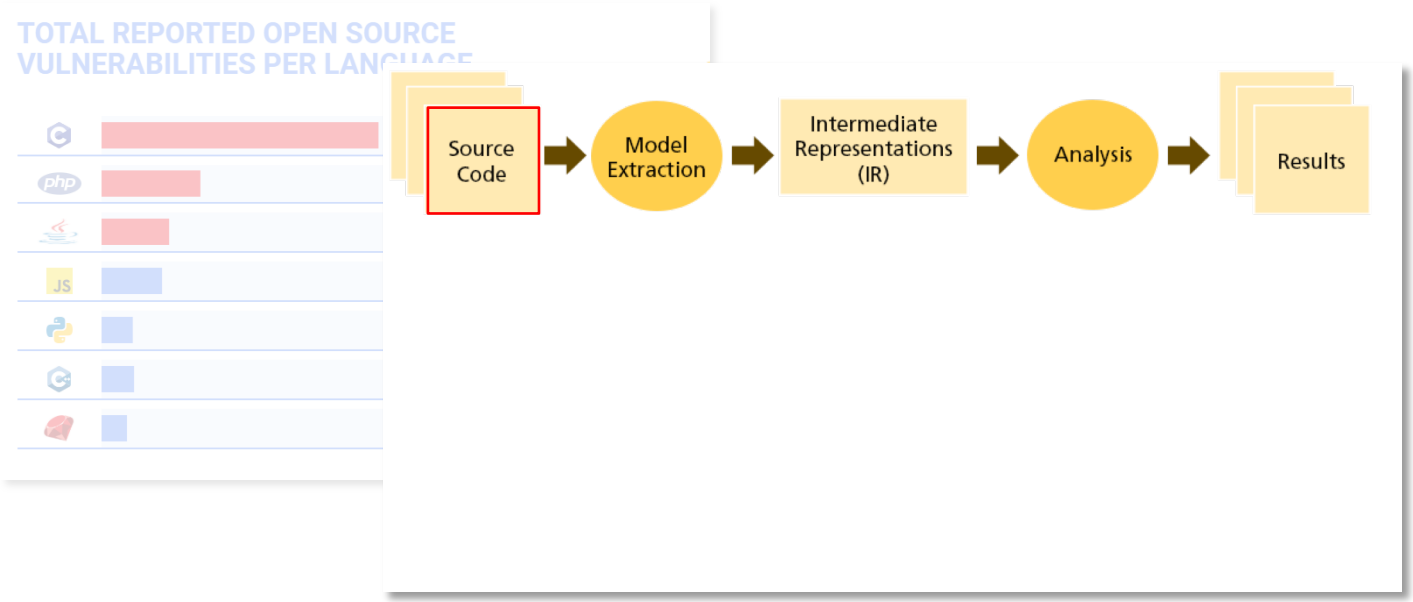


# Motivation



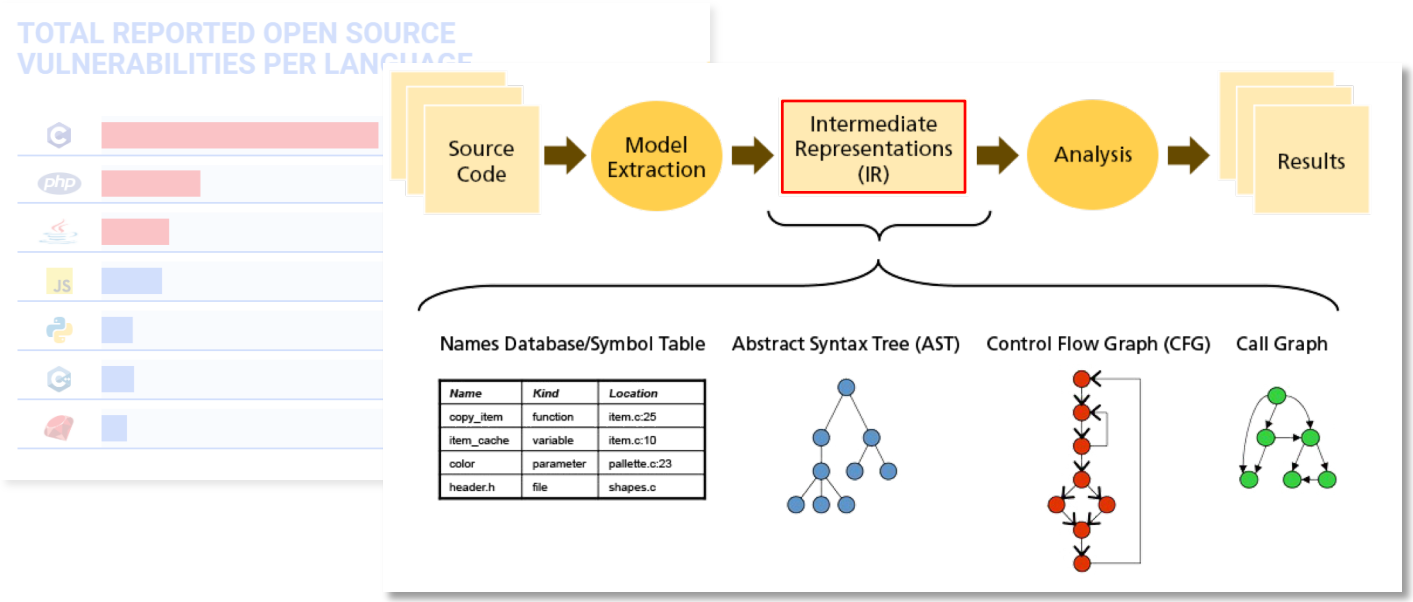

https://www.mend.io/resources/blog/is-one-programming-language-more-secure/

# Static Application Security Testing (SAST)



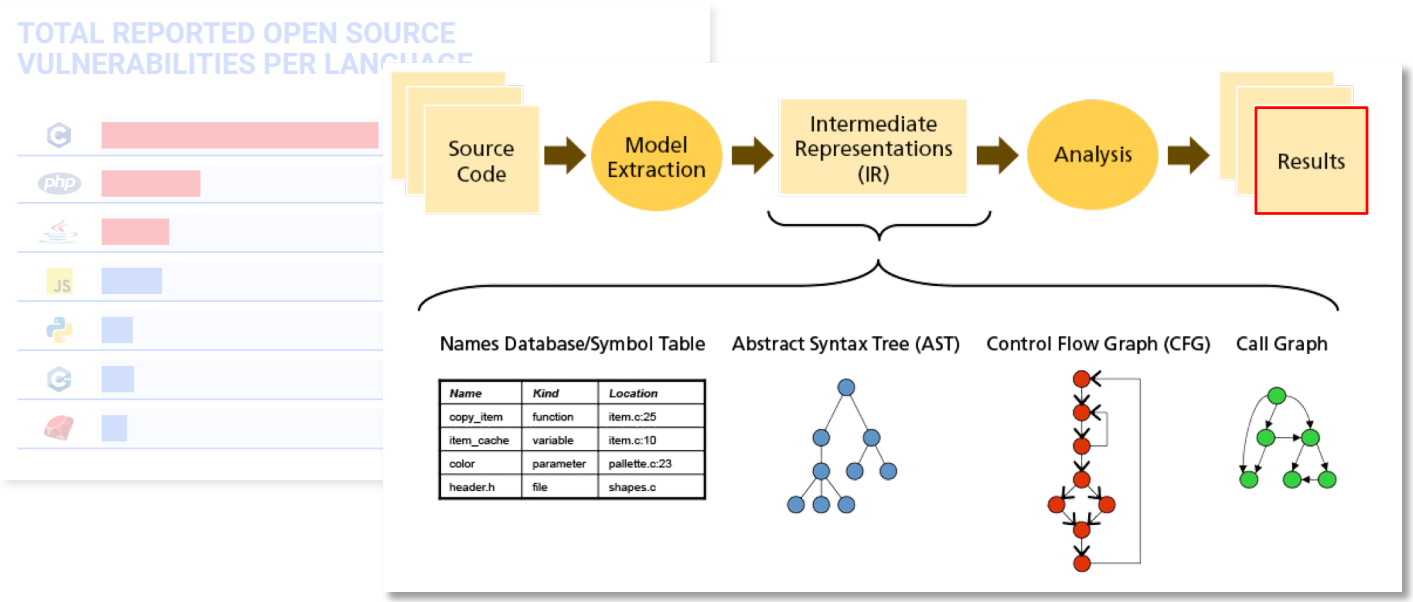

https://www.verifysoft.com/de_grammatech_how_static_analysis_works.html

# Static Application Security Testing (SAST)



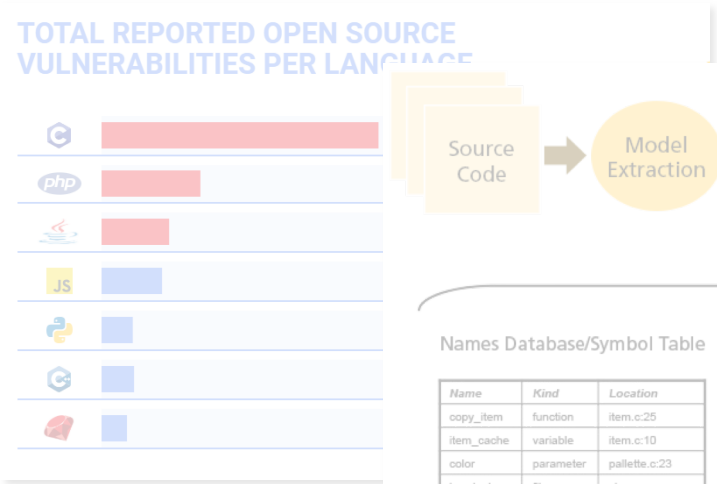

https://www.verifysoft.com/de_grammatech_how_static_analysis_works.html

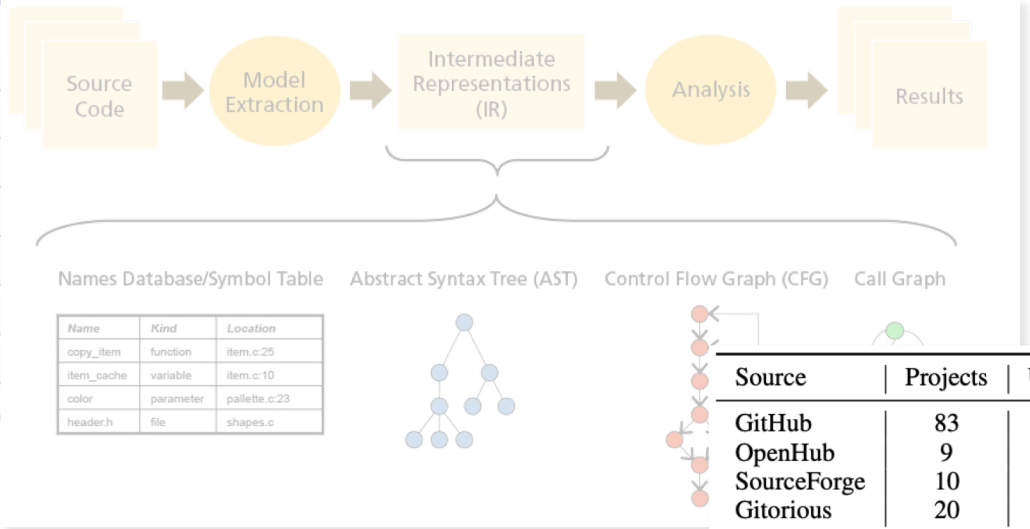
# Static Application Security Testing (SAST)




https://www.verifysoft.com/de_grammatech_how_static_analysis_works.html

# SAST is widely used




| Source | Projects | Use 1 ASAT | Use > 1 ASATs |
|---|---|---|---|
| GitHub | 83 | 34% | 30% |
| OpenHub | 9 | 67% | 22% |
| SourceForge | 10 | 30% | 0% |
| Gitorious | 20 | 30% | 5% |
| Total | 122 | 36% | 23% |

Beller et al. (2016), *Analyzing the State of Static Analysis*

# How effective are SAST tools?

# State-of-Practice & Problem

Existing studies measure the effectiveness of SAST tools mainly on benchmarks with artificial security bugs

- Inserted artificial vulnerabilities are relatively easy to spot, as they are usually inserted in the form of small syntactic code changes
- Evaluations performed on such benchmarks report detection rates of around **80%** – even **100%** for certain types of vulnerabilities – for some of the analyzers studied

Geng et al. (2020), *An Empirical Study on Benchmarks of Artificial Software Vulnerabilities*
Bundt et al. (2021), *Evaluating Synthetic Bugs*

# State-of-Practice & Problem

Existing studies measure the effectiveness of SAST tools mainly on benchmarks with artificial security bugs

- Inserted artificial vulnerabilities are relatively easy to spot, as they are usually inserted in the form of small syntactic code changes
- Evaluations performed on such benchmarks report detection rates of around **80%** – even **100%** for certain types of vulnerabilities – for some of the analyzers studied

**Do the high SAST detection rates for artificial vulnerabilities also hold true for real-world security bugs?**

Geng et al. (2020), *An Empirical Study on Benchmarks of Artificial Software Vulnerabilities*
Bundt et al. (2021), *Evaluating Synthetic Bugs*

# In today‘s talk …

we take a closer look at:

(1) How to automatically evaluate SAST tools against real-world vulnerability benchmarks?

(2) How effective are state-of-the-art SAST tools at detecting vulnerabilities?

# In today‘s talk …

we take a closer look at:

(1) How to automatically evaluate SAST tools against real-world vulnerability benchmarks?

(2) How effective are state-of-the-art SAST tools at detecting vulnerabilities?

# Automated SAST Tool Evaluation

**Why automated evaluation?**

→ Manual analysis:

- Time-consuming: ~200 vulnerabilities, thousands of SAST tool findings
- Subjective: results may be biased towards the reviewer(s)
- Hardly reproducible
- Difficult to extend by further SAST tools

# Automated SAST Tool Evaluation

**Why automated evaluation?**

→ Manual analysis:

- Time-consuming: ~200 vulnerabilities, thousands of SAST tool findings
- Subjective: results may be biased towards the reviewer(s)
- Hardly reproducible
- Difficult to extend by further SAST tools

**Automation = approximation**

→ When do we consider a vulnerability detected by a SAST tool?

# SAST Vulnerability Detection Scenarios

| | | Comparison of vulnerability type? | |
|---|---|---|---|
| | | **No** | **Yes** |
| **# Affected code locations to detect?** | **≥1** | Scenario 1 (S.1-1) | Scenario 2 (S.1-2) |
| | **All** | Scenario 3 (S.2-1) | Scenario 4 (S.2-2) |

# SAST Vulnerability Detection Scenarios

| | | Comparison of vulnerability type? | |
|---|---|---|---|
| | | No | Yes |
| # Affected code locations to detect? | ≥1 | Scenario 1 (S.1-1) | Scenario 2 (S.1-2) |
| | All | Scenario 3 (S.2-1) | Scenario 4 (S.2-2) |

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

# SAST Vulnerability Detection

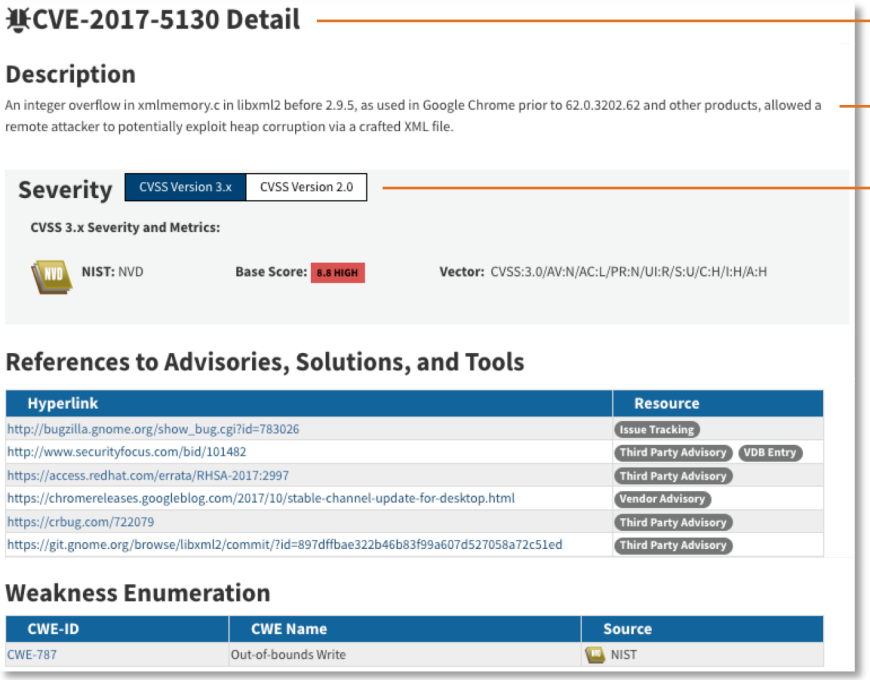
Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

**We need (real-world) ground truth information for both!**

# Ground Truth – **C**ommon **V**ulnerabilities & **E**xposures



## CVE-2017-5130 Detail

### Description

An integer overflow in xmlmemory.c in libxml2 before 2.9.5, as used in Google Chrome prior to 62.0.3202.62 and other products, allowed a remote attacker to potentially exploit heap corruption via a crafted XML file.

### Severity

CVSS Version 3.x | CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

**NIST:** NVD **Base Score:** 8.8 HIGH **Vector:** CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

### References to Advisories, Solutions, and Tools

| Hyperlink | Resource |
|---|---|
| http://bugzilla.gnome.org/show_bug.cgi?id=783026 | Issue Tracking |
| http://www.securityfocus.com/bid/101482 | Third Party Advisory, VDB Entry |
| https://access.redhat.com/errata/RHSA-2017:2997 | Third Party Advisory |
| https://chromereleases.googleblog.com/2017/10/stable-channel-update-for-desktop.html | Vendor Advisory |
| https://crbug.com/722079 | Third Party Advisory |
| https://git.gnome.org/browse/libxml2/commit/?id=897dffbae322b46b83f99a607d527058a72c51ed | Third Party Advisory |

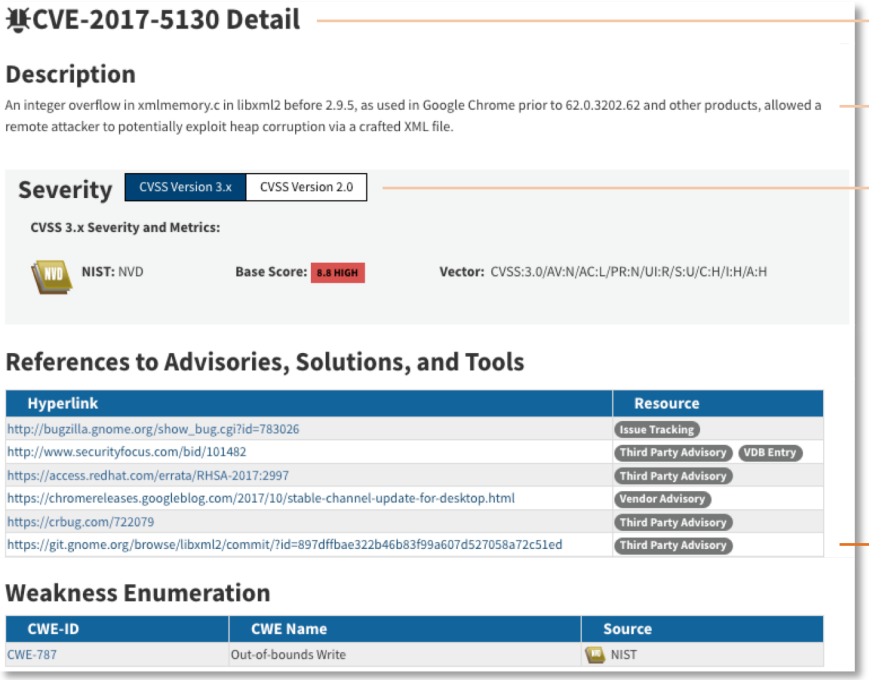
### Weakness Enumeration

| CWE-ID | CWE Name | Source |
|---|---|---|
| CWE-787 | Out-of-bounds Write | NIST |

Vulnerability identifier, description, and severity

https://nvd.nist.gov/vuln/detail/cve-2017-5130

# Ground Truth – **C**ommon **V**ulnerabilities & **E**xposures



## CVE-2017-5130 Detail

### Description

An integer overflow in xmlmemory.c in libxml2 before 2.9.5, as used in Google Chrome prior to 62.0.3202.62 and other products, allowed a remote attacker to potentially exploit heap corruption via a crafted XML file.

### Severity

CVSS Version 3.x | CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

**NIST:** NVD **Base Score:** 8.8 HIGH **Vector:** CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

### References to Advisories, Solutions, and Tools

| Hyperlink | Resource |
|---|---|
| http://bugzilla.gnome.org/show_bug.cgi?id=783026 | Issue Tracking |
| http://www.securityfocus.com/bid/101482 | Third Party Advisory, VDB Entry |
| https://access.redhat.com/errata/RHSA-2017:2997 | Third Party Advisory |
| https://chromereleases.googleblog.com/2017/10/stable-channel-update-for-desktop.html | Vendor Advisory |
| https://crbug.com/722079 | Third Party Advisory |
| https://git.gnome.org/browse/libxml2/commit/?id=897dffbae322b46b83f99a607d527058a72c51ed | Third Party Advisory |

### Weakness Enumeration

| CWE-ID | CWE Name | Source |
|---|---|---|
| CWE-787 | Out-of-bounds Write | NIST |

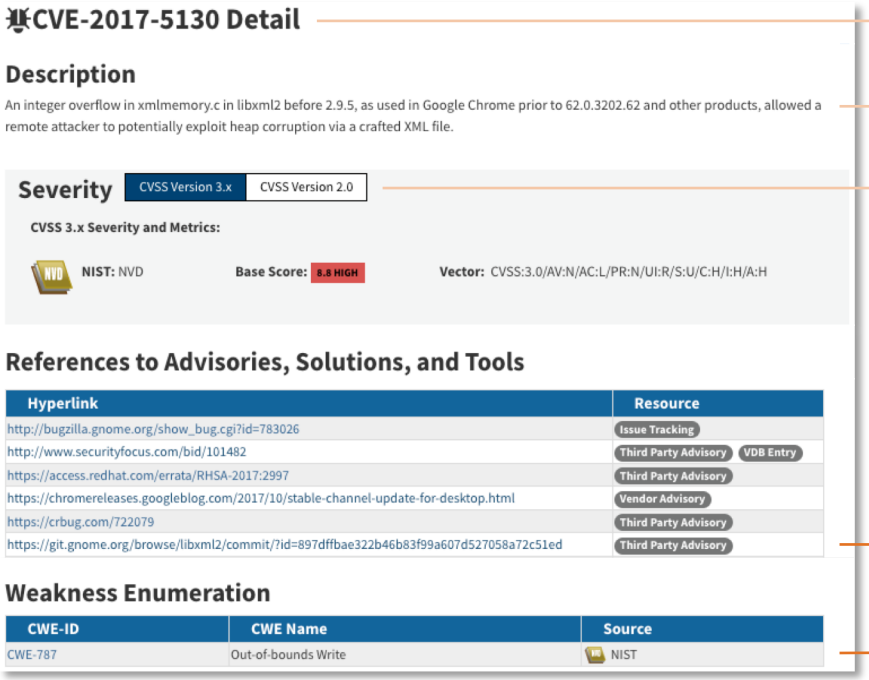
Vulnerability identifier, description, and severity

**(1)** Link to vulnerability patch (GitHub commit)

```
175 + if (size > (MAX_SIZE_T - RESERVE_SIZE)) {
176 +     xmlGenericError(xmlGenericErrorContext,
177 +             "xmlMallocLoc : Unsigned overflow\n");
178 +     xmlMemoryDump();
179 +     return(NULL);
180 + }
181 +
175 182     p = (MEMHDR *) malloc(RESERVE_SIZE+size);
```

https://nvd.nist.gov/vuln/detail/cve-2017-5130

# Ground Truth – **C**ommon **V**ulnerabilities & **E**xposures



**CVE-2017-5130 Detail**

**Description**

An integer overflow in xmlmemory.c in libxml2 before 2.9.5, as used in Google Chrome prior to 62.0.3202.62 and other products, allowed a remote attacker to potentially exploit heap corruption via a crafted XML file.

**Severity** CVSS Version 3.x | CVSS Version 2.0

CVSS 3.x Severity and Metrics:

NIST: NVD Base Score: 8.8 HIGH Vector: CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

**References to Advisories, Solutions, and Tools**

| Hyperlink | Resource |
|---|---|
| http://bugzilla.gnome.org/show_bug.cgi?id=783026 | Issue Tracking |
| http://www.securityfocus.com/bid/101482 | Third Party Advisory, VDB Entry |
| https://access.redhat.com/errata/RHSA-2017:2997 | Third Party Advisory |
| https://chromereleases.googleblog.com/2017/10/stable-channel-update-for-desktop.html | Vendor Advisory |
| https://crbug.com/722079 | Third Party Advisory |
| https://git.gnome.org/browse/libxml2/commit/?id=897dffbae322b46b83f99a607d527058a72c51ed | Third Party Advisory |

**Weakness Enumeration**

| CWE-ID | CWE Name | Source |
|---|---|---|
| CWE-787 | Out-of-bounds Write | NIST |

Vulnerability identifier, description, and severity

**(1)** Link to vulnerability patch (GitHub commit)

```
175 + if (size > (MAX_SIZE_T - RESERVE_SIZE)) {
176 +     xmlGenericError(xmlGenericErrorContext,
177 +             "xmlMallocLoc : Unsigned overflow\n");
178 +     xmlMemoryDump();
179 +     return(NULL);
180 + }
181 +
175 182     p = (MEMHDR *) malloc(RESERVE_SIZE+size);
```

**(2)** Common Weakness Enumeration (CWE), i.e., vulnerability type

https://nvd.nist.gov/vuln/detail/cve-2017-5130

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- **At what abstraction (lines, BBs, or functions) should we check if the marked code matches the vulnerable code?**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- **At what abstraction (lines, BBs, or functions) should we check if the marked code matches the vulnerable code?**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- **How can we (automatically) assess if the vulnerability type output by the SAST tool matches that of the vulnerability?**

# SAST Vulnerability Detection

Levels of approximation:

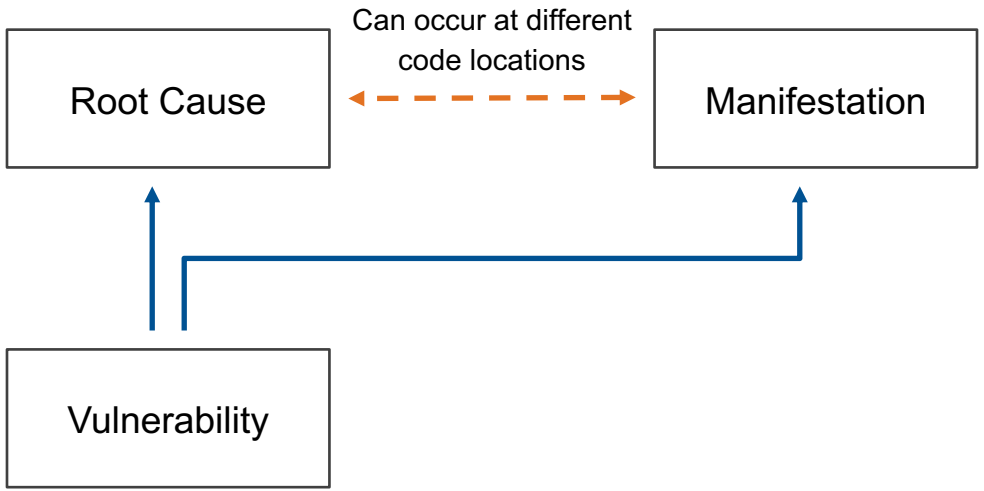
(1) Code location (flagged code ⟺ vulnerable code)

- **At what abstraction (lines, BBs, or functions) should we check if the marked code matches the vulnerable code?**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- **How can we (automatically) assess if the vulnerability type output by the SAST tool matches that of the vulnerability?**

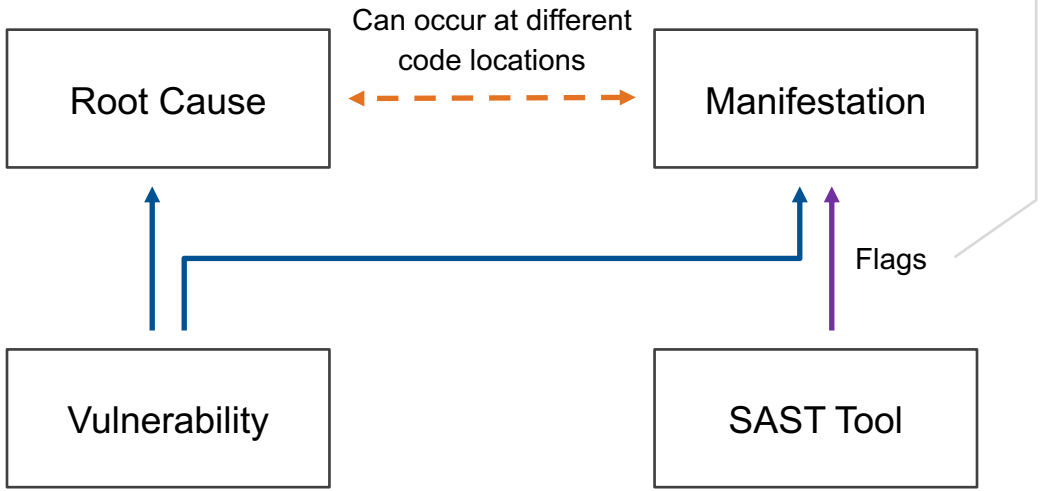
# Root Cause vs. Manifestation



# Root Cause vs. Manifestation

**Root cause analysis is difficult!**

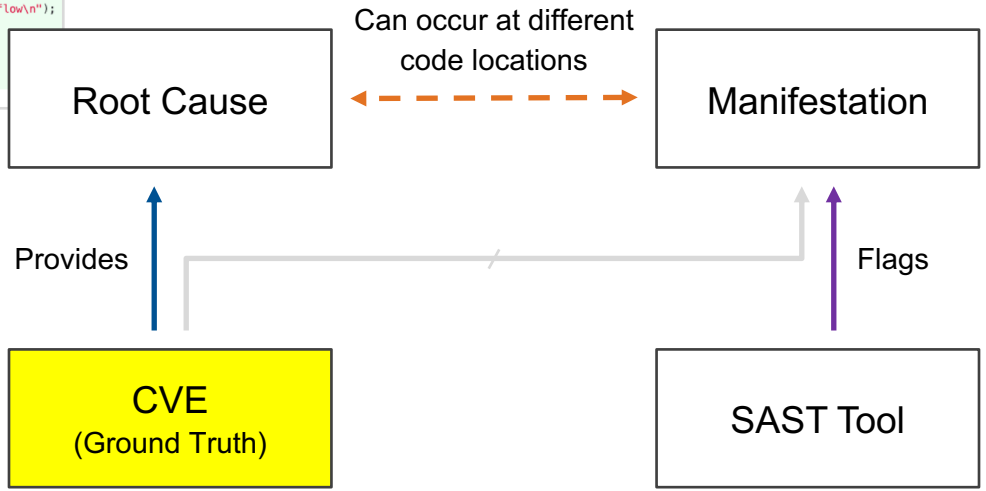
To avoid false positives, SAST tools rather mark the code location(s) where a vulnerability may manifest.



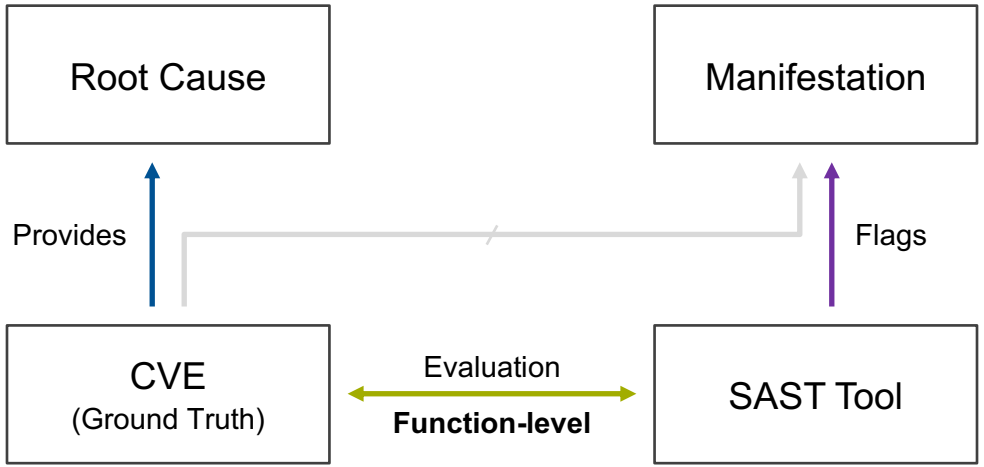
# Root Cause vs. Manifestation

Patch code:

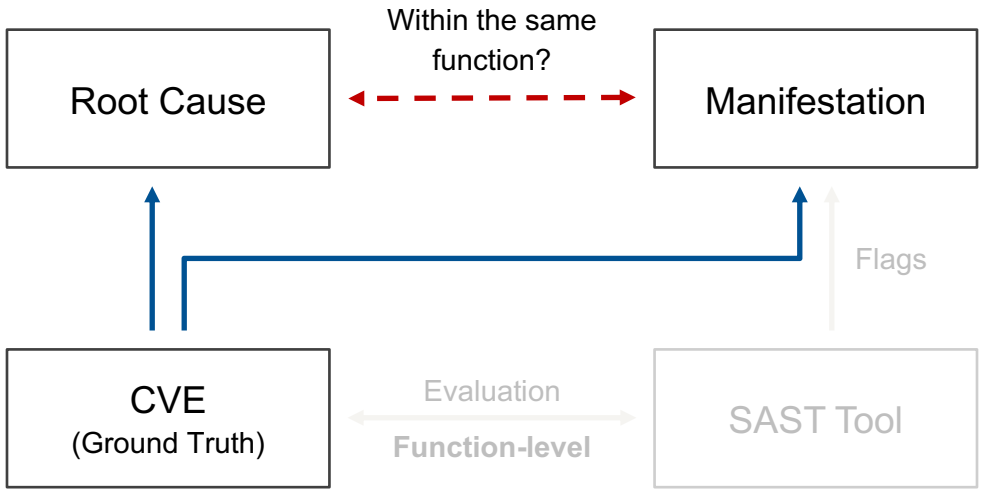
```
175 +   if (size > (MAX_SIZE_T - RESERVE_SIZE)) {
176 +       xmlGenericError(xmlGenericErrorContext,
177 +               "xmlMallocLoc : Unsigned overflow\n");
178 +       xmlMemoryDump();
179 +       return(NULL);
180 +   }
181 +
175 182     p = (MEMHDR *) malloc(RESERVE_SIZE+size);
```



# Detection Code Granularity

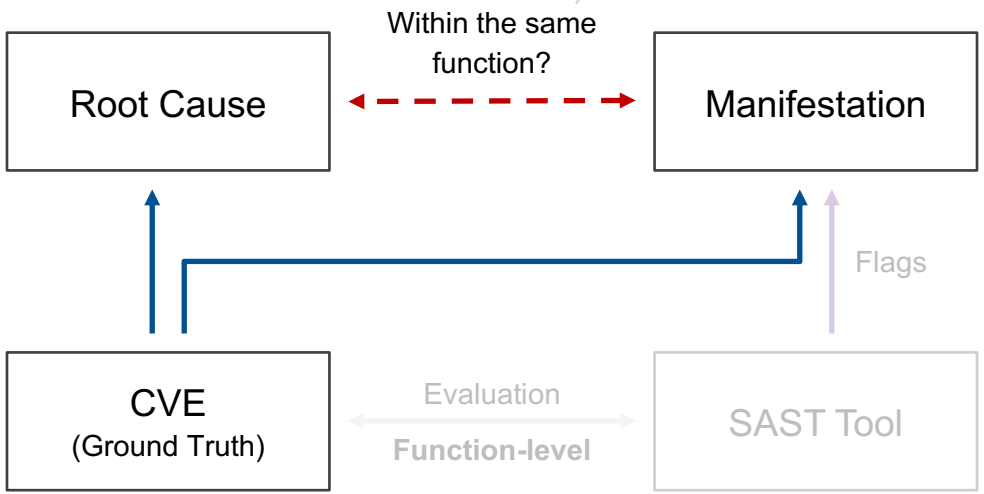


# But …


Within the same function?
Root Cause
Manifestation
Flags
CVE
(Ground Truth)
Evaluation
Function-level
SAST Tool

# But …

**Case study (+50 CVEs):** Root cause and the corresponding manifestation(s) mostly (92%) occur within the same function.



# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- ✓ **Function-level abstraction**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- – **How can we (automatically) assess if the vulnerability type output by the SAST tool matches that of the vulnerability?**

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- ✓ **Function-level abstraction**

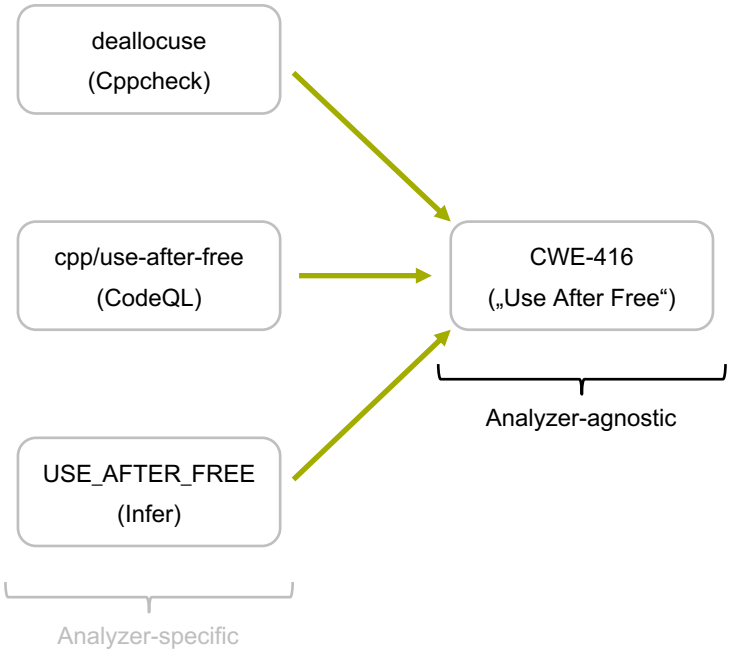
(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- – **How can we (automatically) assess if the vulnerability type output by the SAST tool matches that of the vulnerability?**

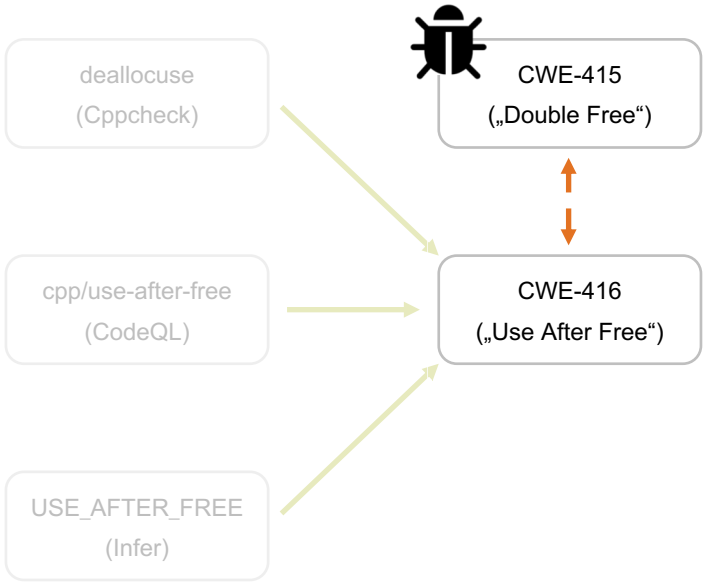
# CWE Mapping …


deallocuse
(Cppcheck)
cpp/use-after-free
(CodeQL)
USE_AFTER_FREE
(Infer)
Analyzer-specific

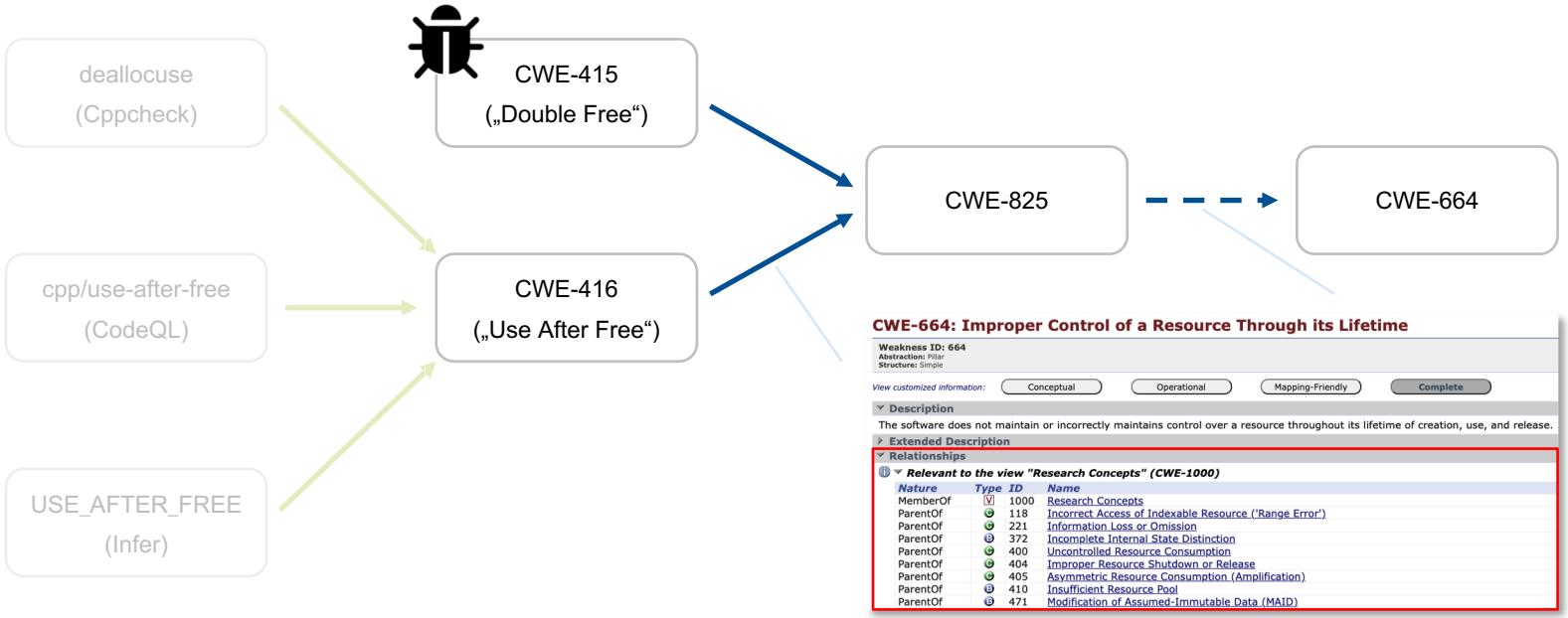
# CWE Mapping …


deallocuse
(Cppcheck)
cpp/use-after-free
(CodeQL)
USE_AFTER_FREE
(Infer)
CWE-416
(„Use After Free“)
Analyzer-agnostic
Analyzer-specific

# CWE Mapping …


deallocuse
(Cppcheck)
CWE-415
(„Double Free“)
cpp/use-after-free
(CodeQL)
CWE-416
(„Use After Free“)
USE_AFTER_FREE
(Infer)

# CWE Mapping and Grouping



# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- ✓ **Function-level abstraction**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- ✓ **CWE mapping and grouping (→ comparison of top-level CWEs, i.e. vulnerability classes)**

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- ✓ **Function-level abstraction**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- ✓ **CWE mapping and grouping (→ comparison of top-level CWEs, i.e. vulnerability classes)**

**Alright, but how „realistic“ is this approximation?**

# SAST Vulnerability Detection

Levels of approximation:

(1) Code location (flagged code ⟺ vulnerable code)

- ✓ **Function-level abstraction**

(2) Vulnerability type (issued vuln. type ⟺ actual vuln. type)

- ✓ **CWE mapping and grouping (→ comparison of top-level CWEs, i.e. vulnerability classes)**

**Alright, but how „realistic“ is this approximation?**

→ **Case study (10 CVEs):** Slight over-approximation of SAST effectiveness when using our automated approach instead of manual analysis!

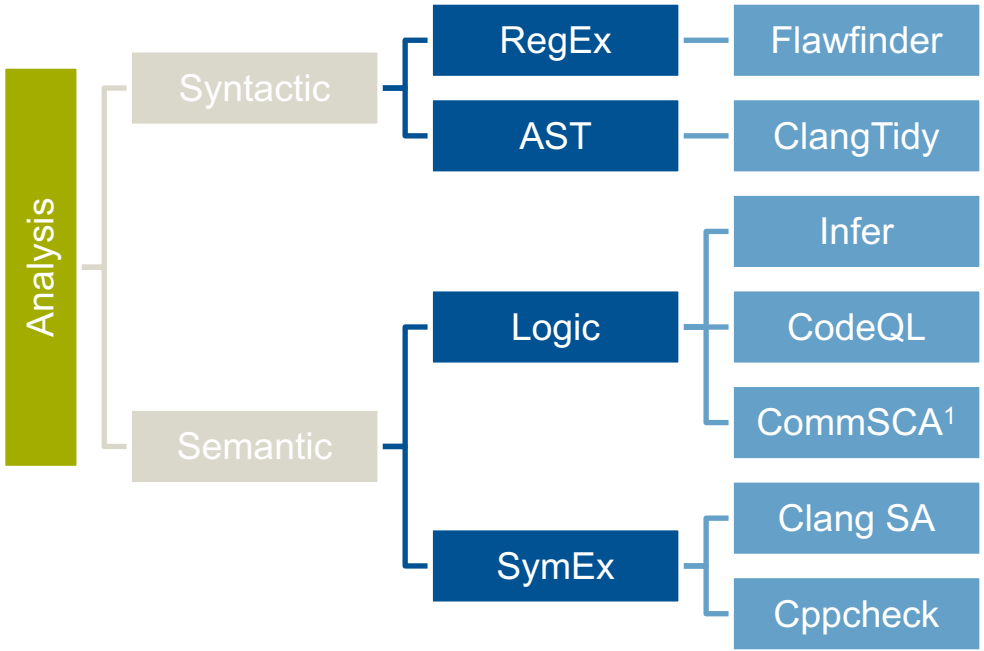
# In today‘s talk …

we take a closer look at:

(1) How to automatically evaluate SAST tools against real-world vulnerability benchmarks?

(2) How effective are state-of-the-art SAST tools at detecting vulnerabilities?

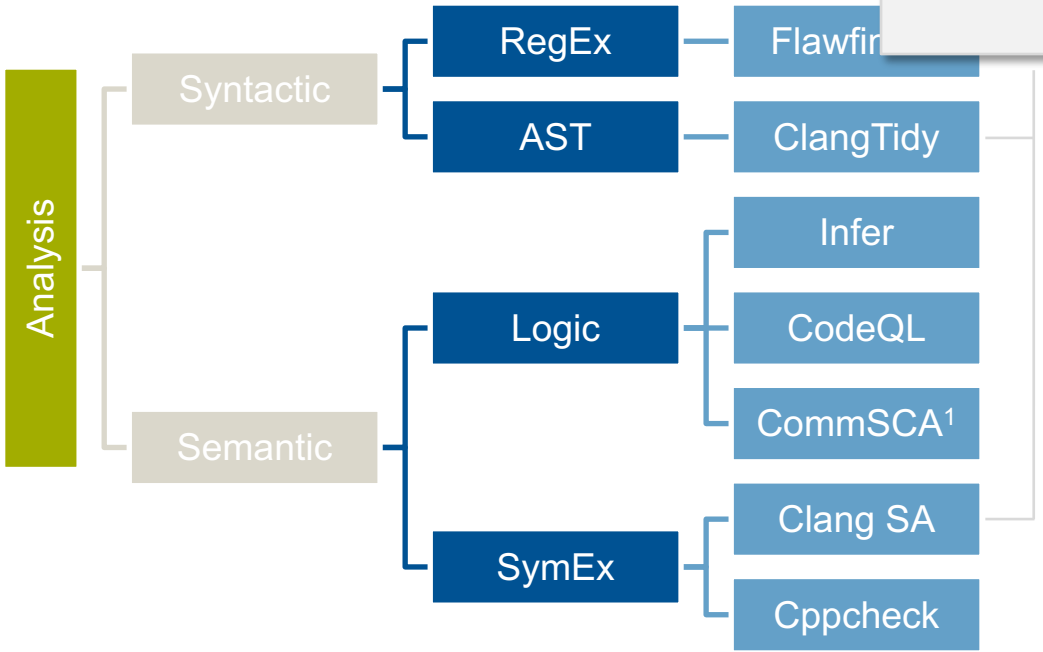
# Selected SAST Tools




[1] Commercial SAST tool

# Selected SAST Tools

We use the static analysis platform **CodeChecker** as „front-end“ for *Clang-Tidy* and *Clang SA*.



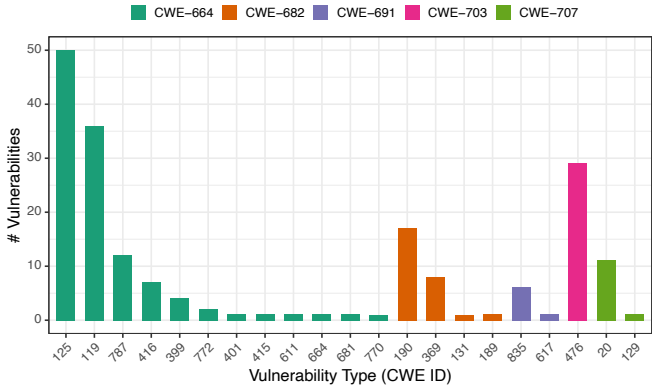

[1] Commercial SAST tool

# Benchmark Dataset

**Subject Programs:**

| Subject | Version | LoC | # Functions | # Vulns. |
|---|---|---|---|---|
| Libpng | 1.6.38 | 10,184 | 398 | 7 |
| LibTIFF | 4.1.0 | 19,527 | 826 | 13 |
| Libxml2 | 2.9.10 | 85,442 | 2,982 | 17 |
| OpenSSL | 3.0.0 | 165,187 | 13,036 | 21 |
| PHP | 8.0.0-dev | 209,407 | 9,145 | 15 |
| Poppler | 0.88.0 | 63,561 | 4,659 | 22 |
| SQLite3 | 3.32.0 | 53,272 | 2,298 | 16 |
| Binutils | 2.29 | 134,767 | 4,071 | 59 |
| FFmpeg | n3.3.2 | 413,353 | 17,788 | 22 |
| Total | | 1,154,700 | 55,203 | 192 |

**Vulnerability Distribution:**




**CWE-664:** Improper Control of a Resource Through its Lifetime (117)

**CWE-682:** Incorrect Calculation (27)

**CWE-691:** Insufficient Control-Flow Management (7)

**CWE-703:** Improper Check or Handling of Exceptional Conditions (29)

**CWE-707:** Improper Neutralization (12)

# RQ.1: SAST Tool Effectiveness


**Analysis time:**

| SAST Tool | PHP | Libxml2 | Libpng |
|---|---|---|---|
| Flawfinder | 00:00:**39** | 00:00:**03** | 00:00:**01** |
| Cppcheck | 00:**25**:04 | 00:**13**:25 | 00:00:**08** |
| Infer | 00:**51**:31 | **13**:05:26 | 00:**04**:05 |
| CodeQL | **02**:25:17 | 00:**55**:47 | 00:**50**:28 |
| CodeChecker | 00:**25**:34 | 00:**10**:47 | 00:**01**:09 |

**FLF:** Flawfinder

**CPC:** Cppcheck

**IFR:** Infer

**CQL:** CodeQL

**CCH:** CodeChecker

**CSA:** CommSCA


**FLF:** Flawfinder

**CPC:** Cppcheck

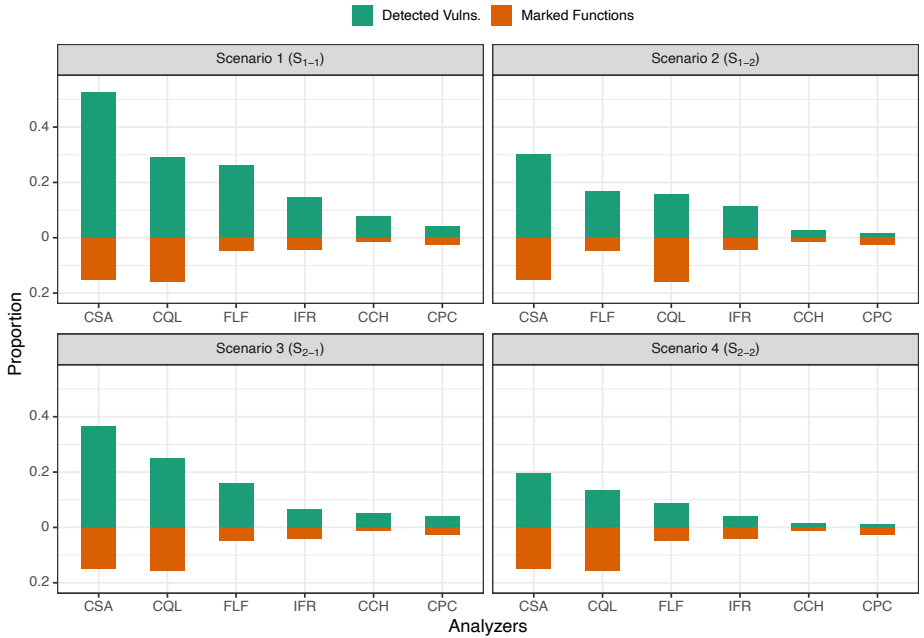
**IFR:** Infer

**CQL:** CodeQL

**CCH:** CodeChecker

**CSA:** CommSCA

# RQ.1: SAST Tool Effectiveness (cont‘d)




**FLF:** Flawfinder

**CPC:** Cppcheck

**IFR:** Infer

**CQL:** CodeQL

**CCH:** CodeChecker

**CSA:** CommSCA

# RQ.1: SAST Tool Effectiveness (cont‘d)


**FLF:** Flawfinder

**CPC:** Cppcheck

**IFR:** Infer

**CQL:** CodeQL

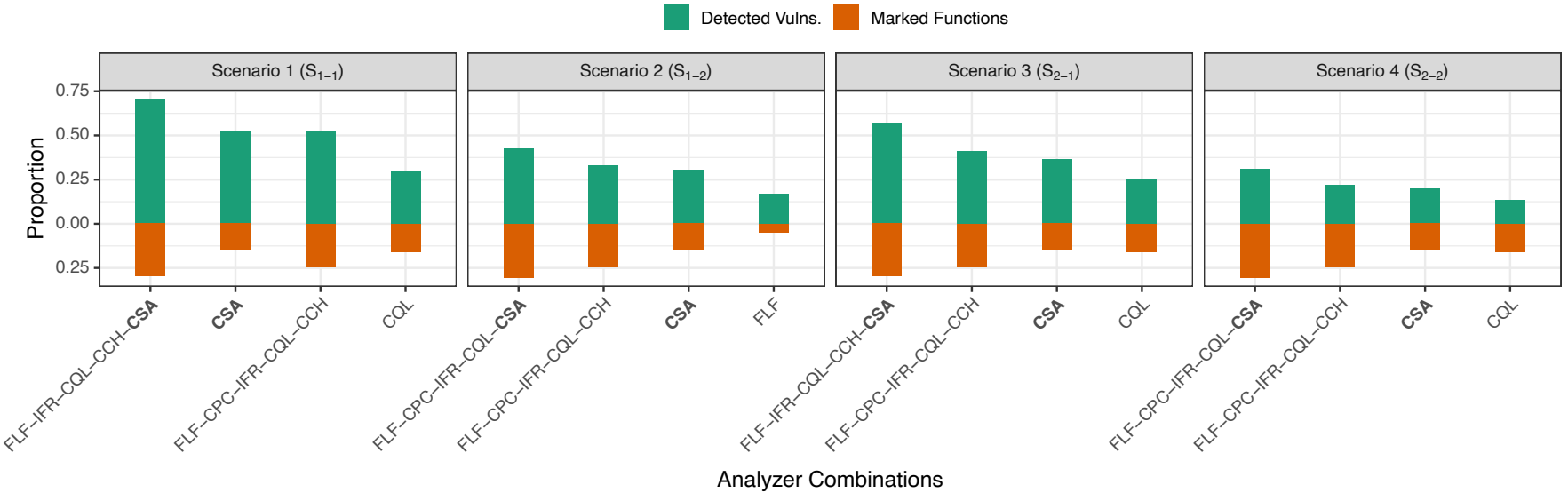
**CCH:** CodeChecker

**CSA:** CommSCA

# RQ.1: Take-Away

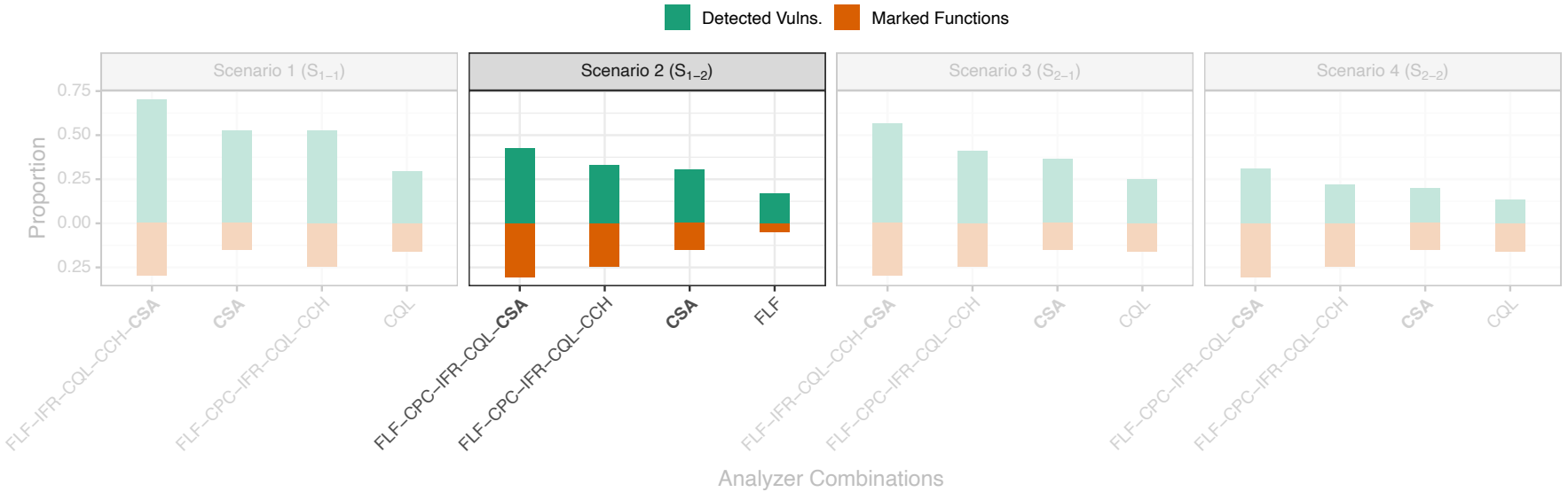
- State-of-the-art C code SAST tools overlook a large number of real-world vulnerabilities
- Even the best tool (CommSCA) misses **47%** (S.1-1), **70%** (S.1-2), **64%** (S.2-1), and **80%** (S.2-2) of the 192 vulnerabilities, while flagging **~15%** of the functions

# RQ.2: Effectiveness Increase by Tool Combinations




**FLF:** Flawfinder, **CPC:** Cppcheck, **IFR:** Infer, **CQL:** CodeQL, **CCH:** CodeChecker, **CSA:** CommSCA

# RQ.2: Effectiveness Increase by Tool Combinations

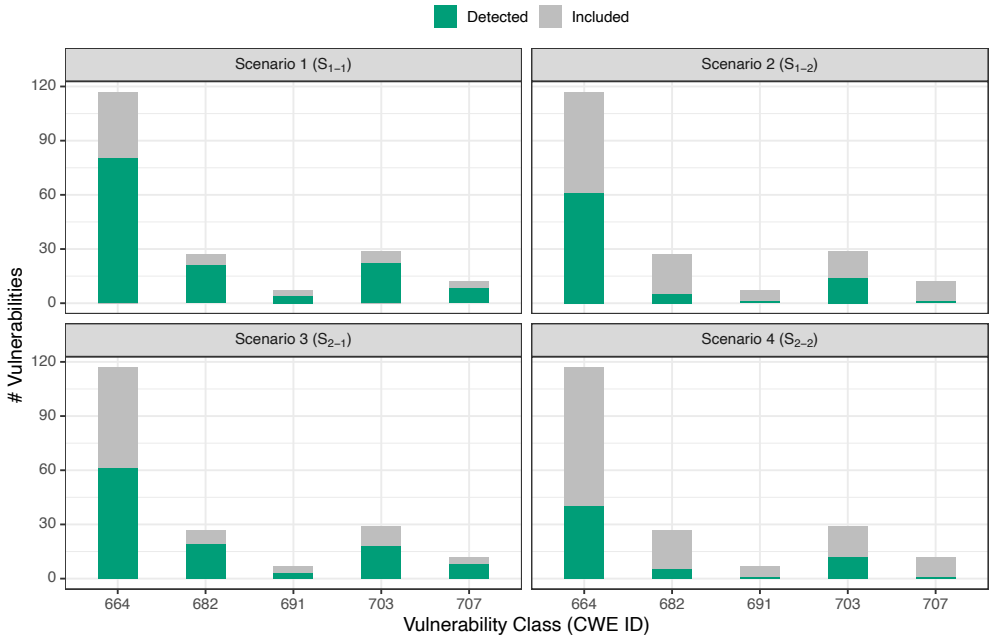



**FLF:** Flawfinder, **CPC:** Cppcheck, **IFR:** Infer, **CQL:** CodeQL, **CCH:** CodeChecker, **CSA:** CommSCA

# RQ.2: Take-Away

- Using multiple SAST tools improves bug detection by **11–34 percentage points** (across the scenarios) compared to a single tool, while flagging **15pp** more functions
- The best combination(s) still miss **30%** (S.1-1), **57%** (S.1-2), **43%** (S.2-1), and **69%** (S.2-2) of the 192 vulnerabilities

# RQ.3: Best vs. Worst Detected Vulnerabilities




**CWE-664:** Improper Control of a Resource Through its Lifetime

**CWE-682:** Incorrect Calculation

**CWE-691:** Insufficient Control-Flow Management

**CWE-703:** Improper Check or Handling of Exceptional Conditions

**CWE-707:** Improper Neutralization

# RQ.3: Best vs. Worst Detected Vulnerabilities


**CWE-664:** Improper Control of a Resource Through its Lifetime

**CWE-682:** Incorrect Calculation

**CWE-691:** Insufficient Control-Flow Management

**CWE-703:** Improper Check or Handling of Exceptional Conditions

**CWE-707:** Improper Neutralization

# RQ.3: Take-Away

- CWE-{664,703} vulnerabilities are more effectively detected than those belonging to CWE-{682,707,691}
- Across the scenarios, **32%–66%** of the 117 CWE-664 vulnerabilities and **24%–59%** of the 29 CWE-703 ones are missed

Check out our dataset:
https://zenodo.org/record/6600197

# Thank you!