CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs towards CWE Detection

Richard A. Dubniczky, Krisztofer Zoltan Horvát, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Norbert Tihanyi

Presented At

TASE 2025

Supported By

ZEISS Digital Innovation

ELTE-OTP Cybersecurity Laboratory

Richard A. Dubniczky

- PhD Student at Eötvös Loránd University (ELTE)
- Lead Security Architect at ZEISS Digital Innovation

richard@dubniczky.com

linkedin.com/in/dubniczky

Structure

1. Motivation & Research Questions
2. Types of Code Weaknesses
3. Types of Static Analysis
4. The CASTLE Benchmark
5. Results
6. Limitations & Conclusion
7. Q&A

Motivation & Problem

- Vulnerabilities in code cause more than a **billion** of users' data to be **exposed** each year. A 1.5 times increase year over year [1].
- Yearly **CVE** releases have tripled since 2017 to around **40,000** in 2024 [2].
- Traditional tools (SAST, formal verification) are widely used to detect faults, but their **effectiveness varies greatly**.
- Uncertainty on what types of **CWE** are covered and how well are they covered by such tools.
- Can **LLMs** be effective tools to augment or replace traditional static analysis?
- How effective are tool **combinations**? What are the best combinations?
- Lack of standardized **metrics** to evaluate single or tool combinations.

[1] Ani Petrosyan, Statista, Annual number of data compromises and individuals impacted in the United States from 2005 to 2024
[2] Kaaviya, cyberpress.org, Over 40,000 CVEs Published in 2024, Marking a 38% Increase from 2023

Research Questions

RQ1: How do state-of-the-art static analysis tools, formal verification methods, and LLM-based approaches compare to effectively detecting C code vulnerabilities?

RQ2: Are combinations of tools more effective than using a single tool?

RQ3: What metrics can reliably demonstrate these differences among various tools?

Bad API Documentation

No Trial

Closed-source tools

Slow Runtime

No CLI interface

Cloud Connection

Restrictive ToS

Errors in code

Trial license

Contact Sales...

No viable dataset

Bugs in C software

Rate limits

API Costs

Mandatory repositories

Custom output formats

Missing Finding Properties

What is a Weakness?

Is there a vulnerability in this snippet?

Yes, technically `printf` can return an error and we did not handle it.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *A = (int*) malloc(sizeof(int) * 10);

    if (A == NULL) {
        printf("Failed to allocate memory.\n");
        return 1;
    }

    for (int i = 1; i <= 4; i++) {
        A[i] = i;
        printf("%d\n", A[i]);
    }

    free(A);
    return 0;
}
```

Common Weakness Enumeration

CWE is a standardized system that categorizes and defines software and hardware security weaknesses to facilitate consistent vulnerability identification, assessment, and mitigation across tools and organizations.

https://cwe.mitre.org/

CWE	Top 25 Rank	Vulnerability Description
CWE-22	5	Improper Limitation of a Pathname to a Restricted Directory
CWE-78	7	Improper Neutralization of Special Elements used in an OS Command
CWE-89	3	Improper Neutralization of Special Elements used in an SQL Command
CWE-125	6	Out-of-bounds Read
CWE-134	12	Use of Externally-Controlled Format String
CWE-190	23	Integer Overflow or Wraparound
CWE-253	-	Incorrect Check of Function Return Value
CWE-327	-	Use of a Broken or Risky Cryptographic Algorithm
CWE-362	-	Concurrent Execution using Shared Resource with Improper Synchronization
CWE-369	23	Divide By Zero
CWE-401	-	Missing Release of Memory after Effective Lifetime
CWE-415	21	Double Free
CWE-416	8	Use After Free
CWE-476	21	NULL Pointer Dereference
CWE-522	14	Insufficiently Protected Credentials
CWE-617	-	Reachable Assertion
CWE-628	-	Function Call with Incorrectly Specified Arguments
CWE-674	24	Uncontrolled Recursion
CWE-761	20	Free of Pointer not at Start of Buffer
CWE-770	24	Allocation of Resources Without Limits or Throttling
CWE-787	2	Out-of-bounds Write
CWE-798	14	Use of Hard-coded Credentials
CWE-822	20	Untrusted Pointer Dereference
CWE-835	24	Loop with Unreachable Exit Condition
CWE-843	-	Access of Resource Using Incompatible Type

Types of Static Analysis

Aspect	Stati Application Security Testing (SAST)	Formal Verification (FV)	Large Language Model (LLM)
Approach	Rule-based pattern matching, data flow analysis, taint analysis	Mathematical proofs of correctness	AI-based pattern recognition and reasoning
Strengths	• Fast runtime • Broad coverage • Easy integration	• Low false positives • Verification guarantees	• High adaptability • Detection without explicit rules
Weaknesses	• High false positive rate • Limited semantic reasoning	• Limited scalability with codebase size • Inability to detect non-formal issues	• Limited context window • Price • Hallucinations and false positives

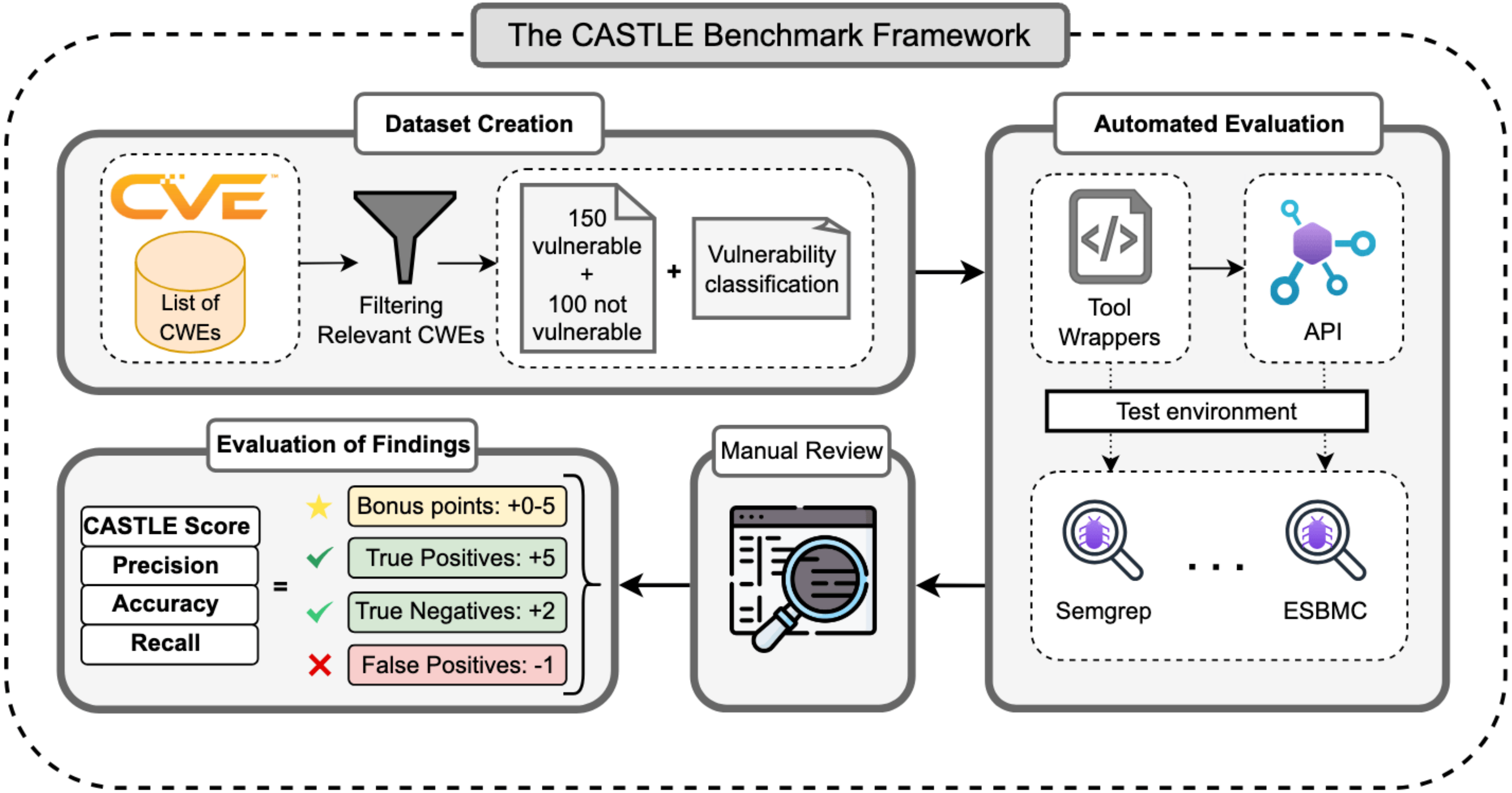
Existing Datasets

Dataset	Size	#Multiple Vuln./File	Vuln. Snippets	Compilable	Granularity	Labelling	Source
Draper [7]	1274k	✔	5.62%	✘	function	Stat	mixed
Big-Vul [8]	264k	✘	100%	✘	function	Patch	real-world
DiverseVul [9]	349k	✘	7.02%	✘	function	Patch	real-world
FormAI-v2 [2]	331k	✔	62.07%	✔	file	FV	AI Gen.
PrimeVul [10]	235k	✘	3%	✘	function	Manual	real-world
SARD [11]	101k	✘	100%	✔	file	B/S/M	mixed
Juliet (C/C++) [12]	64k	✘	100%	✔	file	BDV	synthetic
Devign [13]	28k	✘	46.05%	✘	function	Manual	real-world
REVEAL [14]	23k	✘	9.85%	✘	function	Patch	real-world
CVEfixes [15]	20k	✘	100%	✘	commit	Patch	real-world

What is CASTLE?

CASTLE: CWE Automated Security Testing and Low-level Evaluation

- A benchmarking framework to test static code analyzers
- Consists of 250 compilable, labeled C applications, with 10,000+ LoC
- 150 (60%) of the hand-crafted applications contain one weakness (25 CWEs)
- Wrappers for automated tool evaluation (container, API, report)
- CASTLE Score metric for opinionated evaluation
- CASTLE Combination Score for evaluation of tool combinations



The CASTLE Benchmark Framework
Dataset Creation
CVE
List of CWEs
Filtering Relevant CWEs
150 vulnerable + 100 not vulnerable
+
Vulnerability classification
Automated Evaluation
Tool Wrappers
API
Test environment
Semgrep
...
ESBMC
Manual Review
Evaluation of Findings
CASTLE Score
Precision
Accuracy
Recall
=
Bonus points: +0-5
True Positives: +5
True Negatives: +2
False Positives: -1

Example test (CASTLE-787-1.c)

```c
/*
============================================================================
dataset: CASTLE-Benchmark
name: CASTLE-787-1.c
version: 1.1
compile: gcc CASTLE-787-1.c -o CASTLE-787-1
vulnerable: true
description: Buffer overflow in scanf function.
cwe: 787
============================================================================
*/
#include <stdio.h>

int main( int argc, char *argv[])
{
    char reg_name[12];
    printf("Enter your username:");
    scanf("%s", reg_name); // {!LINE}
    printf("Hello %s.\n",reg_name);
    return 0;
}
```

The CASTLE Metric

Let:

- t: tool (SAST, FV, LLM)
- d: tests
- d_i, weakness in test i
- v_i, correct weakness label for test I
- t_{cwe}, position of a CWE in the MITRE top list

Bonus Point Formula

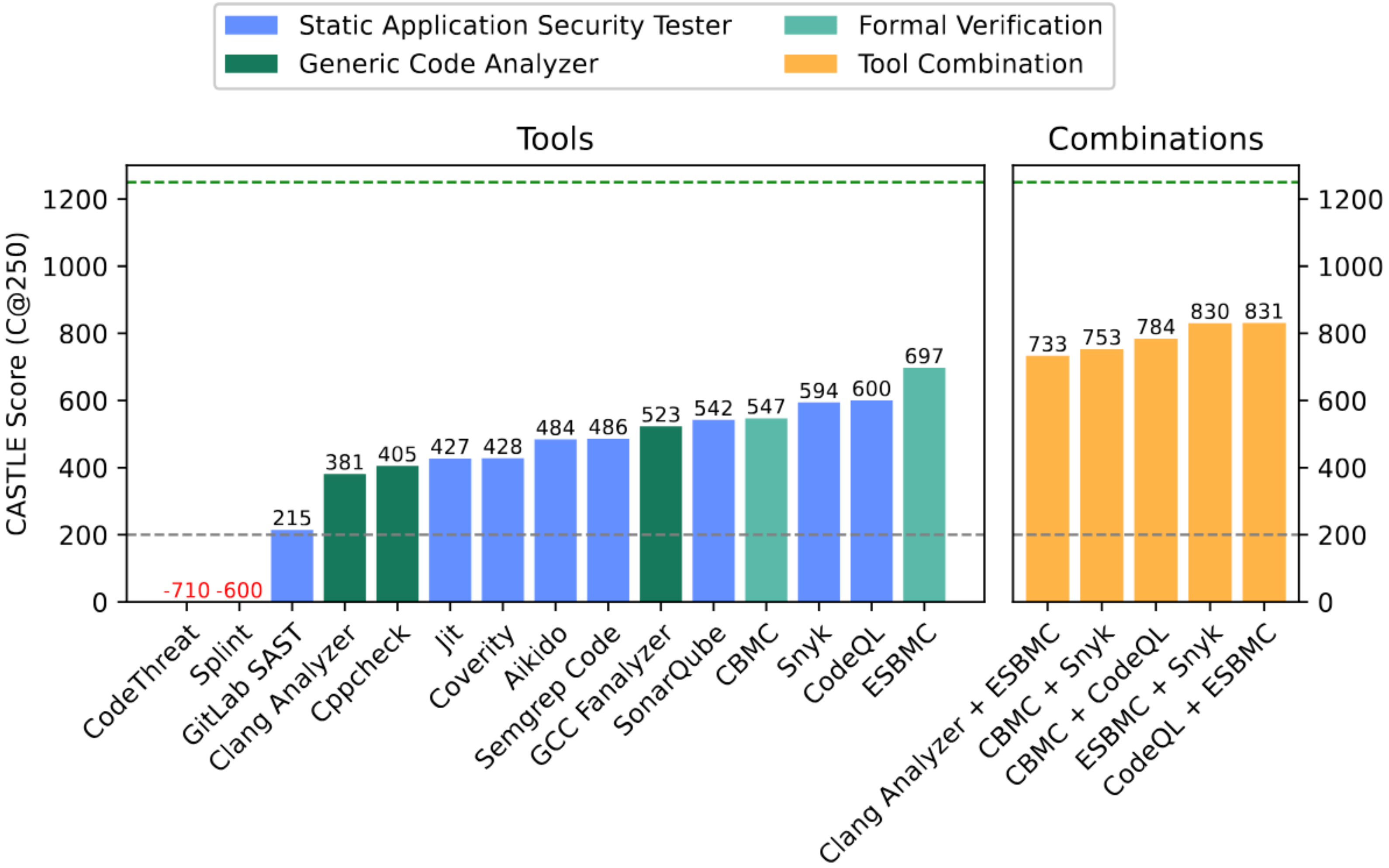
$$B(t_{cwe}) = \begin{cases} b_{\max} - \left\lfloor \frac{S(t_{cwe})-1}{b_{\max}} \right\rfloor, & \text{if } S(t_{cwe}) \leq 25 \\ 0, & \text{otherwise} \end{cases}$$

Tool Score Formula

$$\mathrm{CASTLE}(t, d^n) = \sum_{i=1}^{n} \begin{cases} 5 - (|t(d_i)| - 1) + B(t_{cwe}), & \text{if } v_i \neq \emptyset \wedge v_i \in t(d_i) \\ 2, & \text{if } v_i = \emptyset \wedge t(d_i) = \emptyset \\ -|t(d_i)|, & \text{otherwise} \end{cases}$$

Results

Theoretical maximum: 1250
No findings: 200



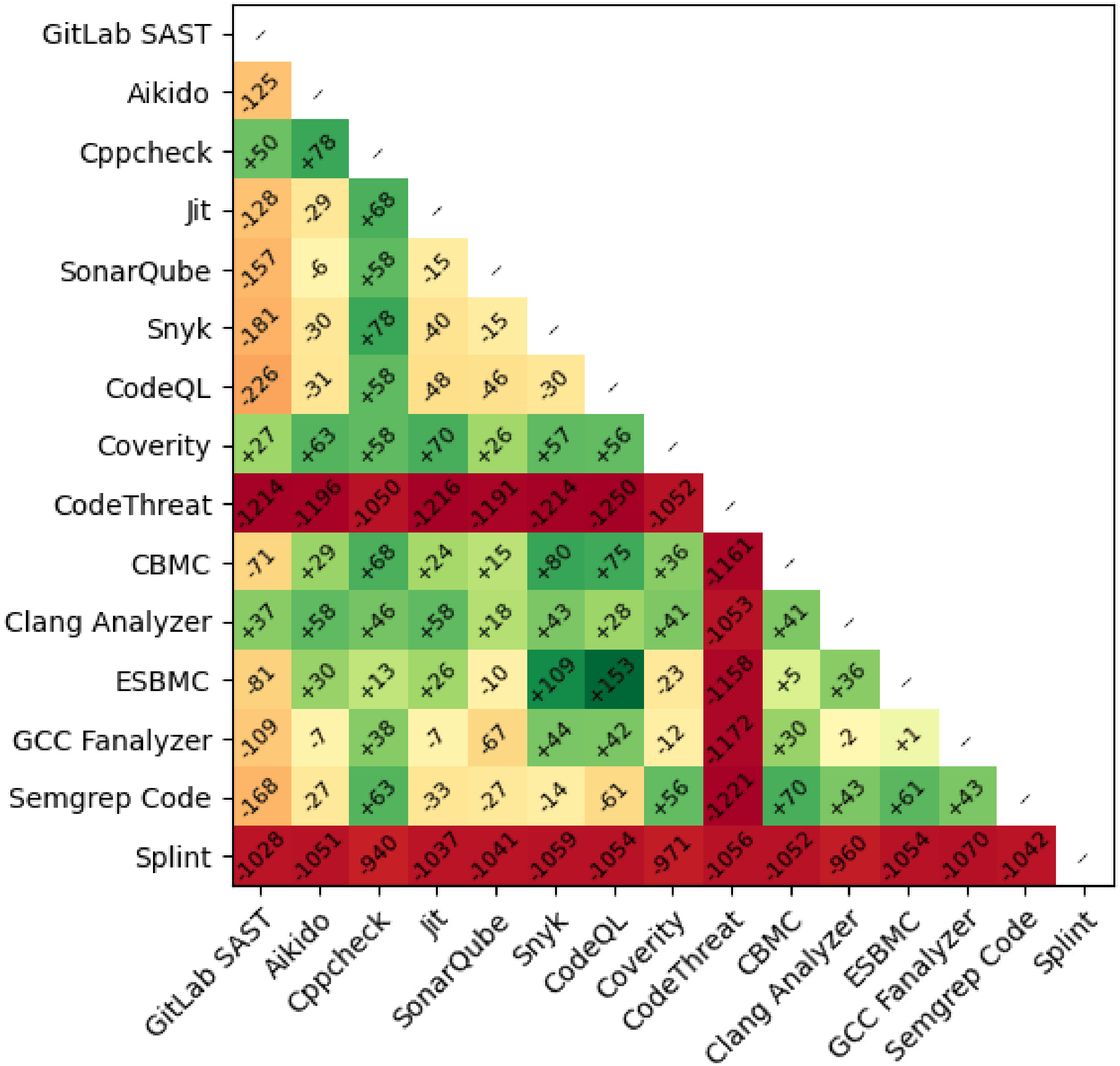
Results

TP = True Positive
TN = True Negative
FP = False Positive
FN = False Negative
P = Precision
R = Recall
A = Accuracy

* All tests were run in February 2025

		Results				Evaluation Metrics			
Name	Version	TP	TN	FP	FN	P	R	A	CASTLE Score
ESBMC	7.8.1	53	99	12	97	82%	35%	58%	697
CodeQL	2.20.1	35	84	43	115	45%	23%	43%	600
Snyk	1.1295.4	30	86	28	120	52%	20%	44%	594
CBMC	5.95.1	41	97	12	109	77%	27%	53%	547
SonarQube	25.3.0	45	73	104	105	30%	30%	36%	542
GCC Fanalyzer	13.3.0	41	81	74	109	36%	27%	40%	523
Semgrep Code	1.110.0	26	76	76	124	26%	17%	34%	486
Aikido	N/A*	12	85	31	138	28%	8%	36%	484
Coverity	2024.12.1	31	87	61	119	34%	21%	40%	428
Jit	N/A*	13	85	58	137	18%	9%	33%	427
Cppcheck	2.13.0	18	100	5	132	78%	12%	46%	405
Clang Analyzer	18.1.3	13	99	2	137	87%	9%	45%	381
GitLab SAST	15.2.1	18	67	259	132	6%	12%	18%	215
Splint	3.1.2	23	36	1029	127	2%	15%	5%	-600
CodeThreat	N/A*	21	2	1104	129	2%	14%	2%	-710
GPT-o3 Mini	-	121	61	72	29	63%	81%	64%	955
GPT-o1	-	114	66	72	36	61%	76%	62%	930
DeepSeek R1	-	133	43	163	17	45%	89%	49%	888
GPT-4o	-	113	45	141	37	44%	75%	47%	814
QWEN 2.5CI (32B)	-	106	31	226	44	32%	71%	34%	666
GPT-4o Mini	-	117	27	276	33	30%	78%	32%	663
Falcon 3 (7B)	-	36	76	70	114	34%	24%	38%	557
Mistral Ins. (7B)	-	54	23	218	96	20%	36%	20%	344
Gemma 2 (9B)	-	42	42	288	108	13%	28%	18%	301
LLAMA 3.1 (8B)	-	56	22	374	94	13%	37%	14%	245

Tool Combination Deltas



Limitations

- Microbenchmark focuses on a single vulnerability in an isolated context
- Small file size: real-world applications are longer than 42 lines
- Long-term “overfitting”: vendors may focus on specific vulnerabilities present here
- Some deviations between runs, especially for LLMs (<3%)
- This study only focuses on the C language, it is hard to generalize

Conclusion

- **LLMs** show great promise moving forward for security analysis with the best current model achieving the **highest score** of 955 points. However, they won't perform as good for larger code bases.
- **FV** methods provide the **most consistent findings** on average, but they are much more limited on the types of issues they find.
- **SAST** tools have a **high false-positive** rate, but they find most higher-level issues coming from semantic issues.
- Tool combinations can be better than a single one if chosen correcylt, out highest tested was 24% better
- The **CASTLE Score** provides a clear and comparable measure of single and combined tool performance.

Future Work

- Extend the code dataset for other common languages (Python, JavaScript, …)
- Introduce random noise with correct functions into the tests to make ”overfitting” more difficult
- Introduction of complex vulnerability chains
- Testing LLMs using RAG
- Testing LLM refining of static analyzer outputs

Thank you for your attention!

Questions?

Contact:

richard@dubniczky.com

linkedin.com/in/dubniczky

Repository:

https://github.com/CASTLE-Benchmark