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

Richard A. Dubniczky⋆ [1], Krisztofer Zoltan Horvát [1], Tamás Bisztray [2,3], Mohamed Amine Ferrag [4], Lucas C. Cordeiro [5,6], and Norbert Tihanyi [1,7]

[1] Eötvös Loránd University (ELTE), Budapest, Hungary
[2] University of Oslo, Oslo, Norway
[3] Cyentific AS, Oslo, Norway
[4] Guelma University, Guelma, Algeria
[5] The University of Manchester, Manchester, UK
[6] Federal University of Amazonas, Manaus, Brazil
[7] Technology Innovation Institute (TII), Abu Dhabi, UAE

**Abstract.** Identifying vulnerabilities in source code is crucial, especially in critical software components. Existing methods such as static analysis, dynamic analysis, formal verification, and recently Large Language Models are widely used to detect security flaws. This paper introduces CASTLE (CWE Automated Security Testing and Low-Level Evaluation), a benchmarking framework for evaluating the vulnerability detection capabilities of different methods. We assess 13 static analysis tools, 10 LLMs, and 2 formal verification tools using a hand-crafted dataset of 250 micro-benchmark programs covering 25 common CWEs. We propose the CASTLE Score, a novel evaluation metric for fair comparison. Our results reveal key differences: ESBMC (a formal verification tool) minimizes false positives but struggles with vulnerabilities beyond model checking, such as weak cryptography or SQL injection. Static analyzers suffer from high false positives, increasing manual validation efforts for developers. LLMs perform exceptionally well in the CASTLE dataset when identifying vulnerabilities in small code snippets. However, their accuracy declines, and hallucinations increase as the code size grows. These results suggest that LLMs could play a pivotal role in future security solutions, particularly within code completion frameworks, where they can provide real-time guidance to prevent vulnerabilities. The dataset is accessible at `https://github.com/CASTLE-Benchmark`.

**Keywords:** Security · Static Code Analysis · Security Analysis · Generative AI · Large Language Models

## 1 Introduction

Rapid advancements in artificial intelligence (AI) have sparked both excitement and concern about the future of traditional software engineering. For instance,

---

⋆ corresponding author richard@dubniczky.com

Meta's recent announcement that AI could soon replace many software engineering roles highlights a shifting landscape in code development [1]. While AI-driven code generation offers remarkable efficiency, a study by Tihanyi et al. found that all examined *Large Language Models (LLMs)* produced vulnerable C code [2]. Similar conclusions have been reached in studies examining other programming languages, such as PHP and Python [3,4]. These large-scale studies consistently indicate that such vulnerabilities arise partly because LLMs lack contextual understanding during the generation process. x'Several studies highlight that once the code is generated, and a vulnerability is identified, LLMs are highly effective at resolving these issues [5,6]. The real challenge is: how do we identify the vulnerabilities? Numerous studies have explored methods for identifying vulnerabilities in large-scale codebases, and various static analysis tools are available on the market. Despite the growing importance of automated software verification, developers and security practitioners lack clear guidance on which tools are most reliable for detecting vulnerabilities in C code. Several interrelated issues contribute to this uncertainty, such as:

1. **Diverse vulnerability types.** Security flaws in C code range from classic memory management issues (e.g., buffer overflows) to subtler logical errors. We need to understand which detection methods can reliably detect different categories.
2. **Emergence of LLMs.** While LLMs exhibit promise in automated code generation, bug fixing, and vulnerability detection, their reliability in different vulnerabilities and coding scenarios is unclear.
3. **Lack of standardized benchmarks.** Existing datasets often contain too many samples with imbalanced CWE representations, and fail to represent the breadth of CWE vulnerabilities. Tools that rely on compilable code—particularly formal verification (FV) methods—are especially disadvantaged without realistic, fully functional programs. To gauge each tool's performance accurately, a benchmark must be rigorously validated, contain clearly labeled vulnerabilities, and support line-level detection granularity.

## 1.1   Motivation

Today, there are two major directions emerging in software engineering, which inspired us to design an entirely new benchmark. Existing benchmarks were no longer adequate to reflect or support these trends. First, code completion and real-time bug detection frameworks are becoming increasingly popular in many *Integrated Development Environments (IDEs)*, as they accelerate application development by automatically correcting common errors and suggesting relevant lines of code. In these scenarios, the focus is typically on small code snippets—usually between 20 and 100 lines—rather than scanning thousands of lines of code. Second, many developers are now utilizing LLMs to assist with various tasks during the software development process. In these cases, LLMs are often tasked with generating simple functions—such as creating a small prime number generator or reading user input to perform basic arithmetic—rather than producing complex systems like full-scale accounting software with tens of thousands of lines of code. In both scenarios, whether code is written by a human in

an IDE or generated by an LLM, the result is typically a small code snippet, and our goal is to accurately identify potential vulnerabilities within that snippet. Given these challenges, a robust and compilable benchmark dataset that accurately captures major CWE vulnerabilities is paramount to answer the following 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?

### 1.2 Main Contributions

Our study holds the following contributions:

- We introduce **CASTLE (CWE Automated Security Testing and Low-Level Evaluation)**, a curated collection of 250 compilable, compact C programs, each containing a single CWE. This benchmark is aimed at enabling direct comparisons among current and future vulnerability scanning tools, including traditional static analyzers, FV techniques, and LLM-based approaches. The small code snippets in the dataset resemble those typically produced by humans or LLMs during the software development lifecycle.
- We conduct a broad comparison of the most widely used static code analyzers and popular LLMs to assess their effectiveness in detecting important vulnerabilities in the C language, using a new metric called **CASTLE Score**, thereby providing crucial insights into their relative strengths and weaknesses.

The rest of this work is structured as follows: Section 2 reviews related literature and outlines the current state of vulnerability scanning tools and AI-based code analysis. Section 3 details the construction of the CASTLE benchmark, including the selection criteria for CWEs and the methodology for creating the curated C programs. Section 4 discusses the results, and presents the experimental setup and comparative analysis of the 13 static code analyzers, 2 format verification tools and 10 LLMs. Section 5 overviews limitations. Finally, Section 6 concludes the paper and outlines potential directions for further research.

## 2 Related Work

Ensuring software correctness, safety, and security is central to software engineering. Examining related literature on the role of AI in software development, most of the existing work and benchmarking approaches focused on testing LLMs' capabilities in producing functionally correct code. However, safety and security are just as important.

## 2.1   Datasets and Benchmarks

Existing vulnerability datasets are frequently used for fine-tuning machine learning models, yet they exhibit several shortcomings that make them unsuitable for comprehensive benchmarking. First, many datasets offer imbalanced representations of CWE categories, failing to provide adequate test coverage of certain vulnerability types. Second, an extreme or uneven distribution of vulnerable versus non-vulnerable samples either hinders accurate false-positive evaluation (when nearly all samples are vulnerable) or fails to capture diverse false-negative scenarios (when some vulnerability types remain underrepresented).

Table 1: C/C++ Datasets for Vulnerability Detection

| 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 |

Legend: **Patch**: GitHub Commits Patching a Vulnerability, **Stat**: Static Analyzer, **BDV**: By Design Vulnerable, **FV**: Formal Verification with ESBMC, **Manual**: Manual Labeling by Human Experts

Furthermore, a key challenge is that many popular datasets lack compilable programs, making it impossible to meaningfully assess formal verification tools such as the *Efficient SMT-based Context-Bounded Model Checker* (ESBMC) [16]. In datasets like SARD [11], which includes the Juliet [12] test cases and 45,437 C samples mapped to CWE categories, many files exceed 3,000 lines of code. This introduces three key constraints:

1. Large token sizes impose high computational costs on LLM-based approaches and limit the use of smaller-parameter models;
2. The complexity and volume of large files can overwhelm formal verification tools, dramatically increasing runtime and impeding direct comparisons with other analyzers;
3. The code samples differ significantly from the small snippets typically generated by LLMs or written by humans within an IDE framework.

Another example is FormAI, a large-scale dataset labeled using ESBMC itself. As a result, it excludes crucial vulnerability classes, such as cross-site scripting (XSS), SQL injection or OS command injection, which exceed the capabilities of current FV tools. One more important point to highlight: most well-known datasets, such as SARD and Juliet, are widely used by tool developers and are also included in LLM training. To avoid bias and to accurately assess the

true capabilities of current tools in identifying vulnerabilities, the creation of a new dataset is essential. This will provide an accurate snapshot of the current strengths and weaknesses of various tools.

## 2.2 The CASTLE Benchmark

CASTLE provides a collection of compilable code snippets, deliberately crafted to cover major CWEs while minimizing the number of queries required for effective analysis. This design enables the straightforward deployment of LLM-based methods and traditional static analyzers with specialized wrappers, facilitating rapid, automated evaluation across various tools. Additionally, the newly introduced *CASTLE score* provides a more detailed comparative metric than conventional pass/fail assessments, allowing for clearer differentiation of subtle performance variations among state-of-the-art tools. The CASTLE dataset balances vulnerable and non-vulnerable samples, permitting more robust evaluations of false positives and negatives.

## 2.3 Traditional Vulnerability Scanning Overview

Traditional approaches have long relied on static analysis methods, such as pattern matching, data flow analysis, and taint analysis, as well as dynamic analysis techniques like fuzz testing [17]. Likewise, *Formal Verification (FV)* methods [18], including *Bounded Model Checking (BMC)* [19] and theorem proving, are widely employed to detect security flaws such as buffer overflows. The NIST-led Static Analysis Tool Exposition (SATE) [20,21] provided large-scale evaluations on open-source code, confirming that while these scanners could spot certain weaknesses, no single method excelled across all vulnerability types.

Academic and industrial benchmarks reveal similar shortcomings. Early work by Wilander and Kamkar [22] showed that five tools missed most C function vulnerabilities and produced many false positives, a trend later echoed by Emanuelsson and Nilsson [23]. Johns and Jodeit [24] demonstrated synthetic benchmarks to distinguish genuine alerts from false alarms, while Bennett [25] reported detection rates of 11.2%–26.5% for standard SAST tools, improved to 44.7% by augmenting them with enhanced Semgrep rules.

## 2.4 LLM-Based Vulnerability Detection

Recent years have witnessed a growing interest in using LLMs for automated vulnerability detection [26,27,28]. Although these models are often praised for handling diverse code repositories, they primarily rely on pattern-based sequence learning rather than (neuro-)symbolic reasoning. As a result, LLMs can detect certain coding flaws effectively, yet they remain susceptible to overlooking complex or context-dependent vulnerabilities. Recent developments, particularly in decoder-only models such as OpenAI's ChatGPT and Meta's Code Llama, highlight a shift in how researchers and practitioners approach vulnerability detection. Their larger context window and on-demand text generation facilitate powerful few-shot or prompt-based strategies that, for specific benchmarks, surpass

classical fine-tuned detectors. For instance, properly designed chain-of-thought prompts have been reported to increase F1 scores on real-world vulnerabilities by providing step-by-step guidance for analyzing the code [4]. Vulnerability detectors typically leverage transformer-based code models trained on massive code corpora spanning multiple languages. These training datasets frequently include insecure code, which can lead to biases or even issues such as *model collapse* [29]. Broadly, transformer models are categorized into three groups [30]:

1. **Encoder-Only**: Used for classification tasks. Early work on vulnerability detection often fine-tuned these models to label code snippets as "vulnerable" or "safe." They generally require full retraining for each new task.
2. **Encoder–Decoder**: Useful for sequence-to-sequence tasks, such as code summarization or refactoring, but they can also be adapted for classification.
3. **Decoder-Only**: Increasingly favored due to large context windows and flexible in-context learning. These models can be prompted to identify vulnerabilities (and sometimes even propose potential fixes) without parameter updates, relying on the knowledge captured during pre-training.

The trend toward decoder-only architectures aligns with industry practices, where state-of-the-art LLMs (e.g., GPT-4) are often served via specialized prompts rather than exhaustive retraining. This approach leverages *in-context learning*, enabling the model to understand and analyze security issues on demand. Carefully constructed prompts—such as chain-of-thought instructions—can improve detection accuracy by guiding the model's attention toward specific code patterns or CWE categories [4]. Existing work indicates that LLM-based solutions can outperform traditional static analyzers on well-defined benchmarks [27,28,26]. However, these improvements do not translate uniformly across all vulnerability types, and use cases: LLMs often fail at detecting nuanced, multi-function flaws or to interpret extensive code segments.

## 3    Methodology

This section overviews the dataset creation process and introduces our research's newly developed evaluation metrics. Figure 1 provides a visual overview of the dataset creation and testing framework.

### 3.1    Dataset

The CASTLE dataset comprises 250 small programs in C, each crafted manually by cybersecurity experts. It encompasses 25 distinct CWEs, with 10 test cases per CWE (6 vulnerable and 4 non-vulnerable). This balanced distribution facilitates focused assessments of each tool's vulnerability detection capabilities while accurately measuring false positives. In ambiguous cases, experts selected a higher-level CWE category or iteratively refined the test until only the most relevant CWE remained. Each program was required to compile without errors, although compiler warnings were permitted. All benchmarks were written in C
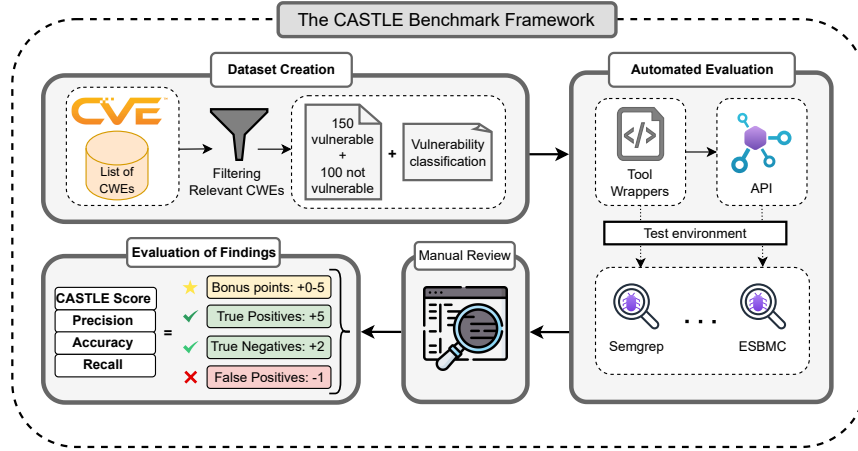
Fig. 1: The CASTLE Benchmark Framework.

and selected for their capacity to accommodate a wide range of vulnerability types, including intricate memory management issues. Furthermore, each test case was restricted to a single file (with optional external libraries) and designed to contain exactly one or zero vulnerabilities. This structure simplifies the identification of vulnerabilities and helps prevent confusion when validating false positives.

When incorporating the system prompt alongside the source code, the total input tokens across the dataset amount to approximately 115,620 tokens using the *cl100k_base* encoding scheme. This total reflects the resource considerations required when running evaluations with token-sensitive language models. The dataset was intentionally capped at 250 benchmarks to make thorough manual verification feasible. This rigorous verification process is indispensable for detecting false positives and confirming line-level detections. Moreover, this selective approach supports the cost-effective evaluation of computationally intensive tools, including advanced LLMs (e.g., GPT-o1, GPT-o3, DeepSeek R1).

The benchmarks exhibit substantial variability in complexity. Code lengths range from 7 to 164 lines, yielding $10,392$ lines (an average of 42 lines). Each includes 1–8 functions (2.2 on average), with cyclomatic complexity values spanning 1–29 (mean 6.3). Halstead volumes range from approximately 89.9 to over $5,246.7$, averaging $1,104.8$. This breadth ensures the dataset covers a wide spectrum of vulnerabilities, from lower-level issues (e.g., memory management flaws, race conditions) to higher-level security risks (e.g., command injections, hardcoded credentials). Most CWEs were chosen based on their prevalence in the Top 25 CWEs of 2023–2024. Each test underwent iterative validation by human experts to ensure overall quality and reliability. Table 2 provides a comprehensive list of the included CWEs.

| 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 |

Table 2: CWEs in the benchmark mapped to MITRE's 2024 Top 25 list [31].

### 3.2   Test Format and Wrappers

Each test in the dataset comprises two components: a metadata block and the source code. Both are stored in a single file for streamlined development and validation, as illustrated in Listing 1.

The metadata, formatted in YAML, precedes the source code and is removed during preprocessing. All lines containing vulnerabilities are marked using the comment string `// {!LINE}`, ensuring consistent identification across different tools. We note that for LLM evaluation, all side-channel information that could introduce bias is removed during the analysis. Additionally, the metadata specifies the vulnerability's CWE classification and other contextual information. After processing, each test is converted into a JSON-formatted dictionary that includes the code, metadata, and computed software metrics (e.g., cyclomatic complexity, Halstead volume). This unified structure simplifies integration with the various wrappers, facilitating automated execution and standardized result reporting. To ensure a uniform and reproducible evaluation across all tools, we developed custom wrappers that automate installation, configuration, execution, and result retrieval. Each tool was containerized via Docker, alongside its dependencies for freely available static analyzers. We then used Python scripts to run each tool on all test cases, collecting and parsing the results into a standardized report format.

For closed-source solutions such as CodeThreat and Aikido, we uploaded the micro-benchmarks to secure repositories or dashboards accessible via proprietary

**Listing 1** An example of a micro-benchmark illustrating a buffer overflow

```
CASTLE-787-1.c Test Source Code

1   /*
2   =================================================
3   dataset: CASTLE-Benchmark
4   name: CASTLE-787-1.c
5   version: 1.1
6   compile: gcc CASTLE-787-1.c -o CASTLE-787-1
7   vulnerable: true
8   description: Buffer overflow in scanf function copying into a fixed length buffer.
9   cwe: 787
10  =================================================
11  */
12  #include <stdio.h>
13  int main(int argc, char *argv[])
14  {
15      char reg_name[12];
16      printf("Enter your username:");
17      scanf("%s", reg_name); // {!LINE}
18      printf("Hello %s.\n", reg_name);
19      return 0;
20  }
```

APIs. The returned results were automatically parsed, and manual consistency checks were performed to verify alignment between reported findings and the tools' web interfaces.

LLM-based vulnerability detection was driven by a generic script that interacted with standard OpenAI APIs. Each model was prompted to return JSON-formatted detection results. Smaller models (fewer than 6B parameters) often struggled to generate well-structured JSON, suggesting limitations in handling detailed output formats. Additionally, models were sensitive to line-specific detections, occasionally identifying the correct vulnerability but offsetting the line number. We also prompted LLMs to provide the corresponding code lines to address minor positioning errors, allowing minimal adjustments during evaluation.

All wrappers developed for this research are publicly available in the main repository. However, intermediate analysis reports are not provided, as they may include proprietary information protected by the respective tool vendors. Each wrapper saves the results in a custom report format, which is later used to process the results and calculate the metrics for the tools.

### 3.3   The CASTLE Score

In this section, we introduce the *CASTLE* score, a new metric for evaluating the performance of vulnerability detection tools with the CASTLE-Benchmark. The CASTLE score integrates both true- and false-positive rates, awards bonus points for detecting high-impact vulnerabilities (based on the Top 25 CWEs), and rewards correct identification of non-vulnerable code. By incorporating these factors, the metric better captures a tool's overall reliability than standard pass/fail evaluations.

Let $d^n = \{d_1, d_2, \ldots, d_n\}$ denote a dataset of $n \in \mathbb{N}^+$ micro-benchmark tests. Each test $d_i$ targets a specific security weakness (e.g., buffer overflow) or contains no vulnerabilities. Let $v_i$ denote the correct vulnerability label associated with $d_i$. If it does not contain a vulnerability, then $v_i = \emptyset$ For any given tool $t$, let $t(d_i)$ represent the set of vulnerabilities reported by $t$ when analyzing $d_i$.

*Bonus Formula:* Following the Top 25 CWE list released by MITRE [31], let $S : \text{CWE} \rightarrow \{1, 2, \ldots, 25\} \cup \{\infty\}$ be a function that returns the rank of a given weakness if it appears in the top 25 list, with $S(c) = \infty$ assigned to any CWE not in the list. Define $b_{\max} = 5$ as the maximum bonus for detecting a Top-25 CWE. For a found vulnerability labeled cwe $= t_{cwe}$, the bonus $B(t_{cwe})$ is computed as:

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

Thus, a tool detecting a highly ranked CWE (e.g., Top 5) receives the full bonus of 5 points, while lower-ranked CWEs yield a proportionally reduced bonus. CWEs outside the Top 25 list receive no bonus.

*Scoring Formula*: For each test $d_i$, a tool's performance is scored according to whether it correctly identifies the vulnerability or the true negative sample. The final CASTLE score for a tool $t$ over the CASTLE benchmark is:

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

**Interpretation:**

– *Correct Vulnerability Detection (True Positive):* If a sample $(d_i)$ is vulnerable $(v_i \neq \emptyset)$ and the tool detects exactly that vulnerability, the tool scores 5 points plus an additional bonus $B(t_{cwe})$ depending on the CWE's standing in the top 25. However, multiple reported findings $(|t(d_i)| > 1)$ reduce the score by one for each, penalizing extraneous detections.

– *Correct Non-Vulnerability Detection (True Negative):* If the sample is non-vulnerable $(v_i = \emptyset)$ and the tool reports no vulnerabilities, it earns 2 points.

– *All Other Cases.* If the tool misses a vulnerability (failing to report $v_i$), or incorrectly flags any vulnerability (including false positives in a non-vulnerable test), the score is reduced by one for each false-positive finding $(-|t(d_i)|)$. Notably, it does not incur additional penalties if the tool reports nothing on a vulnerable benchmark.

We note that assigning zero points for false negatives does not mean the tool avoids penalty for missing a vulnerability. Instead, the absence of points itself acts as the penalty, indicating that no valid finding was made.

### 3.4   The CASTLE Combination Score

An additional feature of the CASTLE score is its applicability to tool combinations. Specifically, if two or more tools exhibit high overlap in detected CWEs, their combined false positives may outweigh any marginal gain from additional true positives, thus lowering the overall score. Conversely, if tools complement each other's coverage without substantially increasing false-positive rates, their combination can yield higher net performance. To compute the *CASTLE Combination Score*, one considers the union of reported vulnerabilities and awards true positives and true negatives once while aggregating penalties for all false positives. This ensures that overlapping detections do not artificially inflate the combined score and that the negative impact of extraneous findings remains cumulative. The combination score can be calculated for any number of tool combinations.

## 4   Discussion

We evaluated  13  static code analyzers,  2  formal verification tools, and  10 LLMs on the CASTLE benchmark. The results, including the CASTLE Scores, are presented in Table 3.

| Name | Version | Results | | | | Evaluation Metrics | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | 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 |

Legend: **TP** = True Positive; **TN** = True Negative; **FP** = False Positive; **FN** = False Negative;
**P** = Precision; **R** = Recall; **A** = Accuracy;
*\* Online API-based tools with unavailable version information (evaluation date: 02/2025)*

Table 3: The results from 250 C tests and their CASTLE Scores.

Tools and LLMs are distinctly separated, and the reasoning behind this will be discussed in this chapter. The CASTLE Score is designed to provide a balanced assessment of a tool's effectiveness by considering both true and false positives and the severity of vulnerabilities. Consequently, not finding a high-severity vulnerability leads to larger penalties than less impactful ones. A negative CASTLE Score could indicate that the volume of false positives generated by a tool imposes a significant triage burden on developers, outweighing its potential benefits. Overall, both the benchmark dataset and the introduced evaluation metric helped highlight various static analyzers' strengths and weaknesses.

## 4.1   Tool Evaluation on the CASTLE Benchmark

Figure 2 presents the results for tools without LLMs, along with their best-performing combinations.
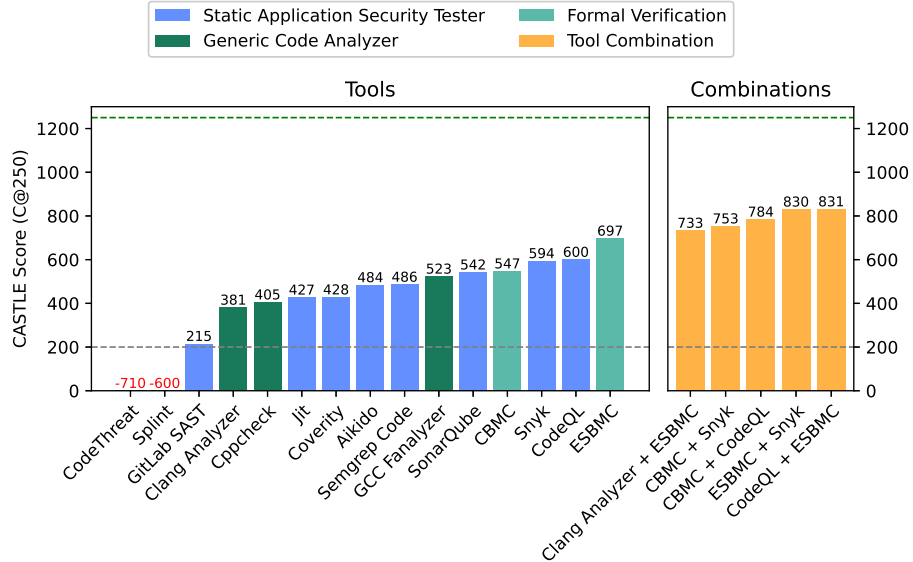


Fig. 2: CASTLE Scores for tools tested on 250 C programs, including the top five tool combinations. Tools reporting no issues score 200 points. The theoretical maximum of a perfect score is 1250 points.

The highest-performing individual tool in our analysis was ESBMC, a formal verification tool. Formal Verification methods have the main disadvantage of being unable to detect non-formal issues, such as SQL Injection, Path traversal, or hard-coded credentials. However, they compensate for this with their low false positive rate. Theoretically, bounded model checkers cannot produce false positives, as they always provide a counterexample to their findings, except in

cases where the tool itself has bugs or reports esoteric scenarios. Both ESBMC and CBMC reported 12 similar but not identical false positives (see Figure 3). These bugs in three categories, some of which we submitted as bug reports to the project [32] [33] [34] [35], fixes are already available for some issues. While this dataset with its short code samples allowed relaxed setting for ESBMC with a longer timeout; `-overflow -no-unwinding-assertions -memory-leak-check -timeout 60 -multi-property -show-stacktrace`, with larger codebases formal verification tools could potentially struggle to finish the verification process in reasonable time, thereby limiting their thoroughness and reliability in giving accurate results. The best-performing SAST tool is CodeQL, which found 23% of the weaknesses in the code (35/150), the highest of the average of 17% among other SASTs. SonarQube found the most, around 30%, but it was dragged down by reporting 2.5 times as many false positives than CodeQL. Several tools, including Clang Analyzer and Cppcheck, displayed high precision (87% and 78% respectively) but struggled with low recall (9% and 12%). This trade-off implies they excel at correctly labeling the few issues they detect, yet they fail to identify a substantial portion of vulnerabilities. Conversely, CodeQL's more balanced approach (45% precision, 23% recall) often provides a more reliable day-to-day detection rate for developers. Splint and CodeThreat generated exceptionally high false positives (1,029 and 1,104, respectively). Their negative CASTLE Scores (-600 and -710) illustrate how overwhelmingly false alerts can erode a tool's practical utility. Although both tools still produced a modest number of true positives, the excessive manual triage effort likely outweighs any marginal benefits for most real-world applications.

Another advantage of the CASTLE score over traditional metrics is that it provides a comparison between using tool combinations. If a pair of tools has a high overlap in the CWEs they can detect, the CASTLE Score of their combination will yield a lower result than the individual tools because of the oversized impact of increasing the rate of false positives. When looking at combination scores, the biggest increase happens with ESBMC and CodeQL, yielding 831 points. This is a 134 point increase over the higher performing ESBMC's base score of 697, and a 19% increase in the effectiveness of using both tools instead of just ESBMC, with a 39% increase above just using CodeQL. This shows that selecting tool combinations with different strengths significantly boosts the efficacy of the static analysis process.

## 4.2   LLM Evaluation on the CASTLE Benchmark

On the CASTLE dataset, LLMs exhibited notably strong performance. In particular, GPT-o3-mini achieved the highest overall score of 955 points, correctly identifying 121 out of the 150 known vulnerabilities. When examining the true positives across different LLM variants, we observed that GPT-4o and GPT-4o-mini generated a similar number of detections than GPT-o1 or GPT-o3-mini for true positives. However, the *reasoning-oriented* models consistently produced fewer false positives, suggesting that their internal steps for "validating" potential vulnerabilities lead to more precise outcomes.
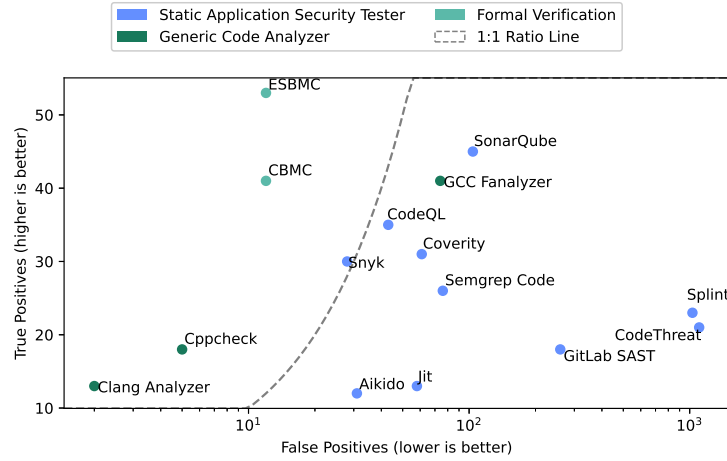
Fig. 3: True Positive vs False Positive rates across tools

Our findings indicate that modern LLMs can pinpoint vulnerabilities in short, self-contained C programs. We conjecture that their neural architectures confer an inherent advantage in pattern recognition, whereas more advanced reasoning models are more effective at minimizing false detections. As a result, LLM-based approaches rival—and often surpass—several classical static analysis tools in detecting common software flaws within compact code segments. However, the next section highlights several limitations and issues for LLMs.

## 5   Limitations

*Microbenchmark Scope:* A fundamental concern with any microbenchmark-based study is its limited scope. Although the CASTLE dataset covers 25 distinct CWEs, each test typically focuses on a single vulnerability in an isolated context. Real-world software often exhibits multi-faceted security flaws spanning tens or hundreds of files. Consequently, tools optimized for detecting specific vulnerabilities may perform artificially well on microbenchmarks while missing complex, cross-file weaknesses that only arise in large-scale applications. Regardless, tools did not perform well on even this small test, indicating that their high false positive rates would be a problem for longer contexts.

*Lack of Large Code Samples:* Preliminary testing with a synthetic 400+ line C program created by merging multiple non-vulnerable test cases, revealed that LLMs tend to report false positives when dealing with larger codebases. Similarly, when one hidden vulnerability was introduced into this combined file, most LLMs failed to detect it reliably, suggesting that these models' effectiveness may taper off with increasing code length. Formal verification approaches also suffer from scalability issues, such as state explosion, and may require lowered bounds that reduce their thoroughness. By contrast, classical static application security

testers (SAST) can handle extensive projects more efficiently, yet their propensity for false positives undercuts overall usefulness in large-scale deployments.

*Potential Overfitting:* Because CASTLE test contents are fixed, tool vendors could theoretically fine-tune their analyzers to excel on known benchmarks, inflating reported accuracy while not generalizing to unseen software. Although this consideration does not impact the integrity of our current study, it underscores the importance of periodically refreshing the dataset or incorporating dynamic test-generation approaches for the future. Furthermore, while repeated evaluations of the same code typically yield consistent results (with observed deviations below 3%), the inherent stochasticity of model-based systems stands in contrast to the deterministic nature of many static analyzers.

## 6   Conclusion

In this study, we introduced the CASTLE benchmark, a curated collection of 250 compilable C micro-benchmarks covering 25 major CWEs. We proposed the CASTLE Score to evaluate diverse vulnerability detection tools, including static analyzers, formal verification methods, and Large Language Models (LLMs). Our work aimed to address the following research questions:

- **RQ1:** *How do state-of-the-art static analysis tools, formal verification methods, and LLM-based approaches compare in effectively detecting vulnerabilities in C code?*
  **Answer:** LLMs exhibit high effectiveness on compact code snippets, with GPT-o3-mini scoring the highest (955 points) by identifying 121 out of 150 vulnerabilities. However, their performance may decline on larger codebases, where false positives increase and hidden vulnerabilities often remain undetected. Static analyzers perform moderately but produce numerous false positives, creating substantial manual triage overhead. Formal verification tools yield minimal false positives within their supported classes (e.g., memory safety) but cannot detect certain higher-level vulnerabilities such as SQL injection, limiting their coverage.
- **RQ2:** *Are combinations of tools more effective than using a single tool?*
  **Answer:** Tool combinations frequently outperform individual tools, particularly when they offset each other's weaknesses. For instance, ESBMC (formal verification) combined with CodeQL achieved the highest two-tool CASTLE Score (831). Although overlapping detections can inflate false positives, well-chosen pairs leverage complementary detection strategies, enhancing overall reliability.
- **RQ3:** *What metrics can reliably demonstrate these differences among various tools?*
  **Answer:** As shown in Table 3, neither precision, accuracy, nor recall could have produced the same results and insights. The CASTLE Score integrates true positives, false positives, and CWE frequency, providing a single, clear measure of tool performance. This setup enables transparent evaluation and straightforward comparisons across diverse methods, even for tool combinations.

**Implications and Future Work.** Although micro-benchmarks efficiently reveal how tools behave on targeted vulnerabilities, they may not reflect the full complexity of production-scale systems. Preliminary experiments indicate that LLMs and formal verification tools both face significant scalability barriers when

analyzing large codebases. Ultimately, the insights gained through CASTLE underscore the importance of selecting and combining tools to fit specific project requirements rather than relying on any single method for comprehensive security assurance.

### 6.1   Conclusion Remark

Finally, we would like to highlight an important point regarding small code snippets. We received feedback from members of the research community expressing concerns that such snippets may not fully reflect real-world scenarios. While it's true that small code snippets may not represent complete, realistic applications, they are sufficient for evaluating the types of vulnerabilities a tool is capable of detecting.

If a tool fails to identify a buffer overflow in a five-line snippet, we cannot reasonably expect it to succeed in detecting the same issue within a larger and more complex codebase. In this sense, the CASTLE-Benchmark provides a valuable theoretical upper bound on a tool's detection capability.

We acknowledge that the rankings presented in Table 3 may vary if these tools are evaluated on larger programs. However, the ability—or inability—of a tool to detect certain vulnerability types will remain consistent. If a tool cannot detect a specific issue in a small snippet, it is unlikely to detect it in a larger context either. For this reason, the CASTLE benchmark is well suited for evaluating methods and tools to determine which are most appropriate for code completion frameworks, where small code snippets are typically analyzed.

## 7   Acknowledgement

**Disclosure of Interests**   The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. G. Marks, "Business tech news: Zuckerberg says ai will replace mid-level engineers soon," *Forbes*, 2025, accessed: 2025-02-03. [Online]. Available: https://www.forbes.com/sites/quickerbettertech/2025/01/26/business-tech-news-zuckerberg-says-ai-will-replace-mid-level-engineers-soon/

2. N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "How secure is ai-generated code: a large-scale comparison of large language models," *Empirical Software Engineering*, vol. 30, no. 2, p. 47, 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10590-1

3. R. Tóth, T. Bisztray, and L. Erdődi, "Llms in web development: Evaluating llm-generated php code unveiling vulnerabilities and limitations," in *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops*, A. Ceccarelli, M. Trapp, A. Bondavalli, E. Schoitsch, B. Gallina, and F. Bitsch, Eds.  Cham: Springer Nature Switzerland, 2024, pp. 425–437.

4. A. Mechri, M. A. Ferrag, and M. Debbah, "Securewen: Leveraging llms for vulnerability detection in python codebases," *Computers & Security*, vol. 148, p. 104151, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824004565

5. M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023.  New York, NY, USA: Association for Computing Machinery, 2023, p. 1646–1656. [Online]. Available: https://doi.org/10.1145/3611643.3613892

6. N. Tihanyi, R. Jain, Y. Charalambous, M. A. Ferrag, Y. Sun, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," 2024. [Online]. Available: https://arxiv.org/abs/2305.14752

7. R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*.  IEEE, 2018, pp. 757–762.

8. J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20.  New York, NY, USA: Association for Computing Machinery, 2020, p. 508–512.

9. Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23.  New York, NY, USA: Association for Computing Machinery, 2023, p. 654–668.

10. Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, " Vulnerability Detection with Code Language Models: How Far Are We? ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*.  Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00038

11. National Institute of Standards and Technology, "Software assurance reference dataset (sard)," 2024, accessed: 2024-11-10. [Online]. Available: https://samate.nist.gov/SARD/

12. N. C. for Assured Software, "Software assurance reference dataset (sard): Juliet c/c++ 1.3," 2024, accessed: November 10, 2024. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

13. Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

14. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2022.

15. G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2021.   New York, NY, USA: Association for Computing Machinery, 2021, p. 30–39.

16. R. S. Menezes, M. Aldughaim, B. Farias, X. Li, E. Manino, F. Shmarov, K. Song, F. Brauße, M. R. Gadelha, N. Tihanyi, K. Korovin, and L. C. Cordeiro, "Esbmc v7.4: Harnessing the power of intervals," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds.   Cham: Springer Nature Switzerland, 2024, pp. 376–380.

17. S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3623375

18. V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

19. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed.   Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

20. V. Okun, R. Gaucher, and P. E. Black, "Static analysis tool exposition (sate) 2008," *NIST Special Publication*, vol. 500, p. 279, 2009.

21. A. Delaitre, P. E. Black, D. Cupif, G. Haben, L. Alex-Kevin, V. Okun, Y. Prono, and A. Delaitre, "Sate vi report: Bug injection and collection," 2023-06-14 04:06:00 2023.

22. J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, 2002, p. 108.

23. P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, 2008.

24. M. Johns and M. Jodeit, "Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools," in *2011 IEEE Fourth international conference on software testing, verification and validation workshops*.   IEEE, 2011, pp. 523–530.

25. G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 614–623.

26. Y. Yang, X. Zhou, R. Mao, J. Xu, L. Yang, Y. Zhang, H. Shen, and H. Zhang, "Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection," *Journal of Systems and Software*, vol. 219, p. 112234, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224002784

27. Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," *arXiv preprint arXiv:2405.17238*, 2024.

28. Y. Lee, S. Jeong, and J. Kim, "Improving llm classification of logical errors by integrating error relationship into prompts," in *International Conference on Intelligent Tutoring Systems*.   Springer, 2024, pp. 91–103.

29. I. Shumailov, Z. Shumaylov, Y. Zhao, N. Papernot, R. Anderson, and Y. Gal, "AI models collapse when trained on recursively generated data," *Nature*, vol. 631, no. 8022, pp. 755–759, 2024. [Online]. Available: https://doi.org/10.1038/s41586-024-07566-y

30. Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "Large language models in software security: A survey of vulnerability detection techniques and insights," *arXiv preprint arXiv:2502.07049*, 2025.

31. MITRE, "2024 CWE Top 25 Most Dangerous Software Weaknesses," 11 2024, available at: `https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html` (Accessed: [Insert Date]).

32. Dr. Norbert Tihanyi, "ESBMC assumption on argv 2312," 2025, accessed: Mar. 8, 2025. [Online]. Available: https://github.com/esbmc/esbmc/issues/2312

33. ——, "ESBMC 7.8 segmentation fault 2236," 2025, accessed: Mar. 8, 2025. [Online]. Available: https://github.com/esbmc/esbmc/issues/2236

34. ——, "GCSE (segmentation fault -PART II) 2235," 2025, accessed: Mar. 8, 2025. [Online]. Available: https://github.com/esbmc/esbmc/issues/2235

35. ——, "Discrepancy in GCSE ( PART II) 2231," 2025, accessed: Mar. 8, 2025. [Online]. Available: https://github.com/esbmc/esbmc/issues/2231