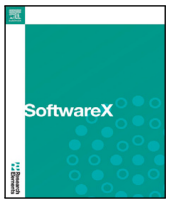
SoftwareX 31 (2025) 102336

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

Unified SAST benchmark: Compare AI-driven and traditional static analyzers

Tamás Viszkok [a,*], Péter Hegedűs [a,b]

[a] University of Szeged, Hungary
[b] FrontEndART Ltd., Szeged, Hungary

ARTICLE INFO

Keywords:
Vulnerability detection
Security
Benchmark
Software analysis
Data visualization
AI vs Static tools

ABSTRACT

In recent years, the growing need for robust security in software development has driven the adoption of static source code analysis tools to detect vulnerabilities early in the software lifecycle. While traditional static analyzers have been widely used, the success of AI in other domains suggests that AI-driven approaches can enhance detection accuracy and reduce false positives. However, a standardized benchmark for comparing AI-based and conventional static analysis tools remains lacking. In this paper, we present an enhanced version of an existing benchmark, extended to support comprehensive comparisons between AI-powered and traditional static code analyzers. The current version includes a dataset of real-world JavaScript vulnerabilities, structured as vulnerable and fixed code pairs—an approach shown to be critical for realistic model evaluation. While the benchmark currently focuses on JavaScript, it is designed to be extensible to other languages in future work. We also introduce a sample static analyzer powered by an AI model to demonstrate practical usage and show how other researchers can easily integrate their own models. Overall, our contributions aim to support reproducible, meaningful evaluation and foster progress toward more secure software development practices.

Code metadata

Current code version	v1.0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00382
Permanent link to Reproducible Capsule	N/A
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	python, Tensorflow, JavaScript, TypeScript, Node.js, React
Compilation requirements, operating environments & dependencies	Docker
If available Link to developer documentation/manual	https://github.com/viszkoktamas/ossf-cve-benchmark/blob/main/README.md
Support email for questions	tviszkok@inf.u-szeged.hu

1. Motivation and significance

The rapid expansion of software development has made security a critical concern, with vulnerabilities posing significant risks to both users and organizations. Static source code analysis enables the detection of security flaws during implementation – before code is complete or executable – allowing vulnerabilities to be addressed early in the development lifecycle. Traditional static application security testing (SAST) tools have long been the standard in this domain [1,2], providing valuable insights. However, these tools often suffer from limitations, such as high false-negative rates [3] and difficulty detecting complex vulnerabilities [4].

Recent advancements in artificial intelligence (AI) have shown promise in improving software security and engineering. AI models can analyze code patterns [5,6], detect vulnerabilities [7,8], and are showing promising results in addressing complex security challenges [9]. As AI adoption grows, researchers increasingly seek reliable ways to compare the effectiveness of AI-driven and conventional SAST tools in

* Corresponding author.
E-mail addresses: tviszkok@inf.u-szeged.hu (T. Viszkok), hpeter@inf.u-szeged.hu (P. Hegedűs).

https://doi.org/10.1016/j.softx.2025.102336
Received 5 June 2025; Received in revised form 25 July 2025; Accepted 27 August 2025
Available online 7 September 2025

realistic security scenarios [10]. Therefore, a benchmark that enables objective, consistent comparisons on real-world examples is essential for meaningful evaluation and advancement.

1.1. Benchmarking datasets

Several publicly available datasets are referred to as "benchmarking datasets" [11,12], but these typically contain only metadata about vulnerabilities, without including the actual source code. They often reference fixing commits in open-source repositories, requiring users to manually retrieve both the fixed and vulnerable code versions. If the vulnerable version is not explicitly provided, users must infer it by identifying the parent of the fixing commit. Once the relevant code is obtained, researchers must manually run tools and implement their own mechanisms for evaluating and comparing results.

This process is time-consuming, inconsistent, and lacks standardization — highlighting the need for dedicated benchmarking frameworks that automate and unify these steps.

1.2. Benchmarking frameworks

Existing benchmarking frameworks also have limitations. For example, the OWASP Benchmark [13], while well-known, is restricted to Java and is therefore unsuitable for our research, which focuses on JavaScript vulnerabilities. Another benchmark, the SAST Benchmark [14], suffers from poor documentation and usability—we were unable to run it, likely due to missing setup instructions and unspecified dependencies. To better understand its functionality, we inspected its source code and configuration files. We found that it runs analyzers on a set of tagged vulnerable and non-vulnerable repositories and collects results into a single HTML file. However, it only lists raw alerts without identifying the exact vulnerability locations, limiting its usefulness for structured evaluation—particularly for AI-based tools.

The most promising solution we found was the OpenSSF CVE Benchmark [15]. It provides a structured, extensible dataset of JavaScript vulnerabilities, where each example consists of a pair of vulnerable and fixed versions of the same repository. This structure is critical for evaluating AI analyzers. Prior research [10] has shown that model performance can drop by over 50% when trained and evaluated on non-overlapping contexts. The ReVeal dataset used in that study, which focuses on C/C++, follows a similar pairing structure. These findings emphasize the importance of using paired examples from the same codebase to assess model generalization realistically.

The benchmark also includes a user-friendly graphical interface that visualizes tool performance using a traffic-light system: green (vulnerability and fix correctly detected), orange (vulnerability detected, fix missed), and red (vulnerability missed entirely). This visual feedback simplifies result interpretation and aids tool comparison.

However, the benchmark was originally designed with traditional SAST tools in mind, which typically report a single line per alert. Our AI-based analyzer, in contrast, often produces multi-line findings. The original benchmark could not handle this and failed during processing. To resolve this, we adapted the framework to support multi-line alerts, preserving its core features while enabling the integration of AI-driven tools. These modifications extend the benchmark's applicability and make it more suitable for current security analysis research.

1.3. Threats to validity

One potential concern with the OpenSSF CVE Benchmark is that it is publicly available. This raises the possibility that static analysis tools – particularly AI-based ones – may have been exposed to the dataset during development, either intentionally or inadvertently. Such exposure could bias evaluation results by artificially inflating a tool's apparent effectiveness. This threat to internal validity should be considered when interpreting benchmarking outcomes.

1.4. Structure of the paper

The remainder of this paper is organized as follows:

- *Section* 2 describes the software, including its architecture, core functionalities, and the integration of traditional and AI-based analyzers.
- *Section* 3 provides illustrative examples to demonstrate key features.
- *Section* 4 discusses the broader impact of the software, including its potential for research and practical adoption.
- *Section* 5 concludes the paper with a summary of contributions and directions for future work.

2. Software description

This enhanced version of the OpenSSF CVE Benchmark addresses a key limitation encountered when using the original framework with AI-based code analyzers: the lack of support for multi-line alerts. The original design assumed that each alerted line would be represented as a separate `Alert` instance. While this structure worked adequately for traditional SAST tools – which typically report a single line per vulnerability – it proved inadequate for AI-driven tools. These often report vulnerabilities at the method or block level, resulting in multi-line alerts. The increased volume and complexity of these alerts caused the original benchmark to crash when processing output from our AI-based tool.

To resolve this, the enhanced version introduced support for representing multiple reported lines within a single `Alert` instance. In parallel, the alert processing and evaluation logic was updated to accommodate multi-line outputs, while preserving the original benchmark's classification scheme: "vulnerability not recognized" (red), "vulnerability recognized but the fix not" (orange), and "vulnerability and fix recognized" (green). These enhancements make the benchmark suitable for evaluating both traditional and AI-based analyzers in a consistent and reliable way.

2.1. Software architecture

Our project consists of two main components: the enhanced benchmark and a sample AI-powered static code analyzer, which demonstrate how traditional and AI-based tools can be integrated into the benchmark.

2.1.1. Architecture of the benchmark (developed by the OpenSSF team)

The evaluation workflow is divided into two phases (see Fig. 1). In the first phase, test cases are loaded from the dataset, and the selected analyzer is run on both the vulnerable and fixed versions of the code. Each tool stores its output in a JSON file. In the second phase, these outputs are compared, and the results are classified using a simple color-coded scheme. The final results are displayed on a visual interface that allows users to quickly assess and compare the performance of different analyzers.

2.1.2. The dataset of the benchmark (collected by the OpenSSF team)

The benchmark is built around a curated dataset of 223 real-world vulnerabilities, each paired with its corresponding fixed version (see Fig. 2). This structure enables tools to be evaluated based on their ability to detect vulnerabilities and recognize fixes. Each dataset entry includes a `CWEs` array listing the relevant Common Weakness Enumerations (CWEs). The dataset spans 38 unique CWE types (see Table 1).

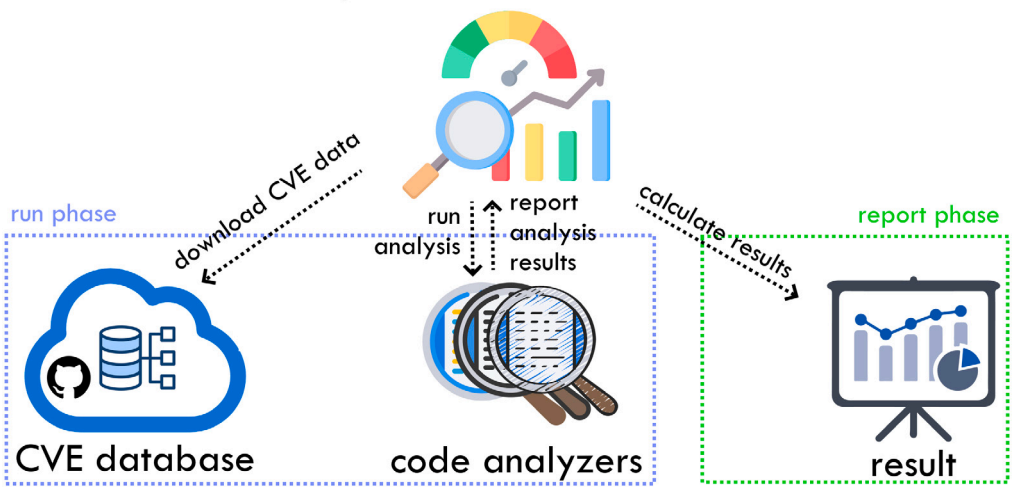


Fig. 1. Architecture of the benchmark.

```
{
    "CVE": "CVE-2017-1000219",
    "state": "PUBLISHED",
    "repository": "https://github.com/KyleRoss/windows-cpu.git",
    "prePatch": {
        "commit": "da656c1a9d5edbf4e8bf0640f349aeb714a4f1a0",
        "weaknesses": [
            {
                "location": {
                    "file": "index.js",
                    "line": 82
                },
                "explanation": "Unsafe shell command constructed from library input"
            }
        ]
    },
    "postPatch": {
        "commit": "b75e19aa2f7459a9506bceb577ba2341fe273117"
    },
    "CWEs": [
        "CWE-078",
        "CWE-088"
    ]
}
```

Fig. 2. An example from the dataset.

Table 1
Distribution of CWEs in the dataset, sorted by frequency.

CWE id	#	CWE id	#	CWE id	#	CWE id	#
CWE-79	83	CWE-36	21	CWE-359	3	CWE-74	1
CWE-116	63	CWE-73	21	CWE-338	2	CWE-399	1
CWE-400	50	CWE-99	21	CWE-200	2	CWE-754	1
CWE-78	48	CWE-20	15	CWE-668	2	CWE-807	1
CWE-94	37	CWE-601	7	CWE-352	2	CWE-290	1
CWE-730	29	CWE-770	4	CWE-829	1	CWE-502	1
CWE-22	29	CWE-89	4	CWE-327	1	CWE-125	1
CWE-88	25	CWE-918	3	CWE-250	1	CWE-126	1
CWE-915	23	CWE-312	3	CWE-444	1		
CWE-23	21	CWE-315	3	CWE-404	1		

2.1.3. Available SAST tools

The original benchmark, developed by the OpenSSF team, includes three integrated SAST tools:

- CodeQL
- ESLint
- NodeJsScan

In addition to these, it features a conceptual baseline tool called *ideal-analysis*, which simulates perfect detection: it reports only true positives and true negatives. This tool complements the benchmark's design, which omits CWEs that are not detected by any selected tool. By including all known vulnerabilities from the dataset, ideal-analysis enables a more comprehensive evaluation of tool coverage and effectiveness.

To expand the benchmark's capabilities, we integrated a fourth traditional analyzer: DevSkim, developed by Microsoft. With this addition, the benchmark now supports the evaluation of four traditional SAST tools alongside the ideal-analysis baseline. The complete set of supported analyzers includes CodeQL, ESLint, NodeJsScan, DevSkim, and ideal-analysis.

2.1.4. The modifications we made on the benchmark — storing multi-line alerts

Originally, the framework was designed to store only a single line number per alert. This design was sufficient for traditional rule-based SAST tools for two main reasons. First, the dataset contains a relatively small number of examples—223 CVEs, each with a corresponding fixed version, resulting in 446 code versions in total. Second, traditional

```
77  findLoad = exports.findLoad = function findLoad(arg, cb) {
78      if(!isFunction(cb)) cb = emptyFn;
79      if(!checkPlatform(cb)) return;
80
81      var cmd = "wmic path Win32_PerfFormattedData_PerfProc_Process get Name,PercentPr
82      exec(cmd, function (error, res, stderr) {
83          if(error !== null || stderr) return cb(error || stderr);
84          if(!res) return cb('Cannot find results for provided arg: ' + arg, { load: 0
85
86          var found = res.replace(/[^\S\n]+/g, ':').replace(/\:\s/g, '|').split('|').f
87              return !!v;
88          }).map(function(v) {
89              var data = v.split(':');
90              return {
91                  pid: +data[0],
92                  process: data[1],
93                  load: +data[2]
94              };
95          });
96
97          var totalLoad = 0;
98
99          found.forEach(function(obj) {
100             totalLoad += obj.load;
101         });
102
103         var output = {
104             load: totalLoad,
105             found: found
106         };
107
108         cb(null, output);
109     });
110 };
```

Fig. 3. Multi-line vulnerability alert reported by CodeQL.

tools typically report one line per alert, which kept the number of alert entries low. Even if a vulnerability spanned multiple lines (see Fig. 3), storing it as multiple alert instances did not cause performance issues.

However, this structure became limiting when integrating AI-based analyzers, which tend to report vulnerabilities at the block or method level. Initially, we attempted to store each reported line as a separate `Alert` instance. This resulted in a large number of alert entries, which caused the framework to fail during report generation, revealing scalability and performance bottlenecks.

To address this, we modified the benchmark to support multi-line alerts by allowing a single `Alert` instance to store a list of affected lines. We updated the JSON output format accordingly and adapted each integrated tool to produce output in this new structure (see Figs. 4 and 5). The benchmark's processing logic was also extended to correctly interpret and load multi-line entries.

Originally, the benchmark required an exact match between the alert line and the vulnerable line from the dataset. We relaxed this condition to accommodate multi-line alerts: an alert is now considered valid if it contains the vulnerable line. For example, as shown in Fig. 3, a CodeQL alert might span lines 82–109, while the dataset marks only line 82 as vulnerable (see Fig. 2). Since line 82 falls within the alert's range, the match is accepted.

These updates resolved the immediate performance issues and made the benchmark more robust. We also introduced additional refinements to improve its scalability and maintain compatibility with tools that produce more granular output.

2.1.5. The modifications we made on the benchmark — simplified execution and tool integration

To simplify setup and usage, we containerized the benchmark using Docker and provided a script that automates the entire workflow. Docker ensures a consistent runtime environment across different systems, while the script handles building and executing the benchmark with minimal user interaction. This removes the need for manual dependency management and reduces setup time, allowing users to focus on evaluation rather than configuration.

We also automated the installation and configuration of the integrated analyzers. Instead of requiring users to install each tool individually and update configuration files manually, the Docker image includes the tools and their setup routines. New analyzers can be added easily by following the existing directory structure and naming conventions, as documented in the project's README.

This simplified execution model improves usability, encourages adoption by lowering the barrier to entry, and helps ensure reproducible results across different environments.

2.1.6. Our AI-driven SAST code analyzer tool

One of the integrated analyzers is our AI-based static code analysis tool (see Fig. 6). It operates as a modular component that can be plugged into the benchmark with minimal configuration. At its core is an AI model that processes source code and predicts the presence of vulnerabilities. The architecture is designed to be flexible—users can replace the underlying model without modifying the rest of the system.

The tool consists of two parts: a Node.js module and a Python module. The Node.js part handles user input and extracts functions from relevant source files. The extracted code is then passed to the Python module, which performs vulnerability prediction. By default, the Python module uses a neural network model, but this can be replaced with any compatible implementation.

The tool accepts either a directory or a list of source files as input. When given a directory, it recursively scans for source files while ignoring paths listed in standard ignore files such as '.gitignore' or '.dockerignore'. Each discovered function is extracted and sent to the AI model for analysis.

In the default setup, source code is vectorized using Microsoft's CodeBERT model [16], and predictions are made by a Keras-based neural network trained on a publicly available JavaScript vulnerability dataset [17] using the Deep-Water Framework [18]. To demonstrate

```
{
  "runs": [
    {
      "CVE": "CVE-2017-1000219",
      "commit": "da656c1a9d5edbf4e8bf0640f349aeb714a4f1a0",
      "toolID": "codeql",
      "status": "SUCCESS",
      "reproduction": "cd /tmp/codeql-run-YvSDpk &&\n /usr/src/app/analyzers,
      "alerts": [
        {
          "location": {
            "file": "index.js",
            "line": 82
          },
          "ruleID": "js/shell-command-constructed-from-input"
        },
        {
          "location": {
            "file": "index.js",
            "line": 83
          },
          "ruleID": "js/shell-command-constructed-from-input"
        },
        {
          "location": {
            "file": "index.js",
            "line": 84
```


Fig. 4. Previous JSON report structure (per-line alerts).


```
{
  "runs": [
    {
      "CVE": "CVE-2017-1000219",
      "commit": "da656c1a9d5edbf4e8bf0640f349aeb714a4f1a0",
      "toolID": "codeql",
      "status": "SUCCESS",
      "reproduction": "cd /tmp/codeql-run-YvSDpk &&\n /usr/src/app/analyzers,
      "alerts": [
        {
          "location": {
            "file": "index.js",
            "lineStart": 82,
            "lineEnd": 109
          },
          "ruleID": "js/shell-command-constructed-from-input"
        }
      ]
    }
  ]
}
```


Fig. 5. Updated JSON report structure (multi-line alerts).

modularity, we also implemented an alternative using a small language model from the Qwen2 family [19] via Huggingface Transformers [20].

These AI models are included for demonstration purposes only. While they do not produce state-of-the-art results, they are lightweight and fast to run. More information on replacing the model can be found in the README files of the project [21] and the analyzer itself [22].

2.1.7. Our AI-driven SAST code analyzer tool — optimization

Most static analysis tools are designed to analyze either entire project directories or individual files (see Figs. 7 and 8). However, the benchmark dataset typically focuses on a small subset of files that contain the actual vulnerabilities and their corresponding fixes.

To improve efficiency and reduce noise, we extended our analyzer to accept an explicit list of files as input, in addition to single files or directories. This allows targeted analysis of only the relevant files, significantly reducing the number of irrelevant alerts and improving runtime performance.

By narrowing the scope of analysis, this optimization not only speeds up execution but also helps produce cleaner, more interpretable results during benchmarking.

2.2. Software functionalities

The main functionalities of the software are:

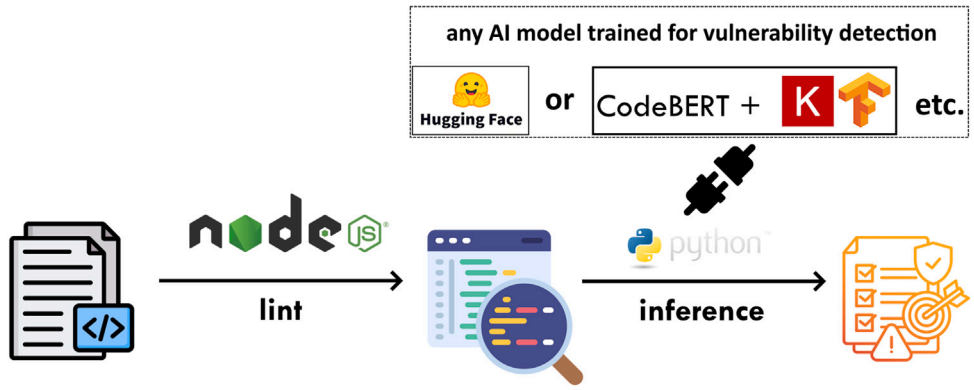


Fig. 6. Architecture of our AI driven code analyzer.

```
Shell

codeql database create [--language=<lang>[,<lang>...]] [--github-auth-stdin] [--github-url=
<url>] [--source-root=<dir>] [--threads=<num>] [--ram=<MB>] [--command=<command>] [--
extractor-option=<extractor-option-name=value>] <options>... -- <database>
```

Fig. 7. CodeQL source directory configuration example.

```
Usage: devskim analyze [arguments] [options]

Arguments:
  -I, --source-code            Required. Path to a directory containing files to scan or a single file to scan.
  -O, --output-file            Optional. Filename for result file, uses stdout if not set.
```

Fig. 8. DevSkim source directory configuration example.

- *Benchmarking SAST tools*: The enhanced framework can now evaluate both traditional and AI-based static analysis tools by running them on pairs of vulnerable and fixed code versions and classifying their outputs using a color-coded scheme.
- *AI-based code analysis*: Our analyzer uses an AI model to identify vulnerabilities in source code. Its modular architecture allows researchers to swap out the model easily, enabling experimentation with different approaches.
- *Dockerized execution*: The entire system runs in a preconfigured Docker environment, eliminating the need to manually install dependencies or manage compatibility issues. This ensures consistent and reproducible evaluation across different machines.

3. Illustrative examples

The original OpenSSF benchmark provides a visual interface for comparing the performance of different static code analyzers. Fig. 9 shows an example of the output after evaluating multiple tools, including our AI-based analyzer (`ossf-sast-ml`), across various vulnerability categories.

Each row in the visualization corresponds to a specific Common Weakness Enumeration (CWE) and aggregates results from multiple vulnerable–fixed code pairs associated with that category. For each code pair, the benchmark checks whether the analyzer correctly identifies the vulnerability in the vulnerable version and whether it recognizes the fix in the patched version.

The results are visualized using horizontal color-coded bars, where each color indicates a specific detection outcome:

- *Green*: The tool detected the vulnerability in the vulnerable version and did not flag it again in the fixed version, indicating it recognized the fix.
- *Orange*: The tool detected the vulnerability in the vulnerable version but still flagged it in the fixed version, meaning it failed to recognize the fix.
- *Red*: The tool failed to detect the vulnerability in the vulnerable version.

The summary row at the top summarizes each tool's overall performance across all tested CVEs, helping users assess both vulnerability detection accuracy and the ability to distinguish fixed code from still-vulnerable code.

This visualization helps researchers and practitioners compare traditional and AI-based tools across real-world vulnerabilities, and it supports reproducible, transparent evaluations at scale.

4. Impact

The enhanced benchmark and the accompanying AI-powered static code analyzer address a growing need for standardized, reproducible comparisons between traditional and AI-based static analysis tools [23]. Additionally, the benchmark can be used independently to evaluate and compare AI models focused on vulnerability detection.

While several existing studies have introduced datasets for evaluating AI-based vulnerability detection tools [10–12], few have provided reusable, framework-based benchmarks tailored to such models. Our work fills this gap by extending an existing benchmark to support AI tool integration and by providing an execution environment

Fig. 9. Results of the benchmark.

that focuses specifically on real-world JavaScript vulnerabilities. Researchers can now systematically test how their models react to real vulnerability–fix pairs and compare them under consistent conditions.

This capability opens new research directions related to model generalization, comparative evaluation strategies, and standardized performance metrics—all essential for advancing the field of AI-based code security analysis.

The Dockerized setup dramatically reduces the barrier to entry for running large-scale evaluations. With just a single command, users can launch the benchmark without manually managing complex dependencies or environment configurations. This design supports developers, educators, and researchers in integrating the tools into their workflows with minimal effort.

Both the benchmark framework and our AI analyzer are open-source and publicly available on GitHub [21,22]. This transparency supports reproducibility, encourages broader adoption, and enables researchers to build upon our work in future studies.

5. Conclusions

In this work, we presented an enhanced benchmarking framework designed to evaluate both traditional and AI-based static source code analyzers in the domain of software security. By building on top of an existing benchmark and extending it to support AI integration, we addressed a growing demand for reproducible and meaningful comparisons in this space.

The benchmark allows researchers to assess how well tools detect and track fixes to real-world JavaScript vulnerabilities. Results are presented through a simple, interpretable visual system that facilitates comparison across tools. In addition, we introduced an AI-powered static analyzer with a modular architecture, enabling easy integration of alternative models for experimentation.

To simplify deployment, the entire system has been containerized using Docker, removing the need for manual setup and ensuring consistency across environments.

Together, these contributions support open and reproducible experimentation and promote further research into both AI-driven and traditional approaches to vulnerability detection.

Future improvements may include extending the dataset with more examples, supporting additional programming languages and vulnerability categories, and expanding the list of integrated tools.

CRediT authorship contribution statement

Tamás Viszkok: Writing – original draft, Visualization, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Péter Hegedűs:** Writing – review & editing, Validation, Supervision, Funding acquisition.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used ChatGPT (GPT-4o) for editorial corrections and proofreading of the manuscript. After using this tool/service, the authors reviewed and edited the content as needed and took full responsibility for the content of the published article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory; and in part by the Project no TKP2021-NVA-09 implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

The work has also received support from the European Union Horizon Program under the grant number 101120393 (Sec4AI4Sec).

References

[1] Beller M, Bholanath R, Mcintosh S, Zaidman A. Analyzing the state of static analysis: A large-scale evaluation in open source software. 2016, http://dx.doi.org/10.1109/SANER.2016.105.

[2] Vassallo C, Panichella S, Palomba F, Proksch S, Gall H, Zaidman A. How developers engage with static analysis tools in different contexts. Empir Softw Eng 2020;25. http://dx.doi.org/10.1007/s10664-019-09750-5.

[3] Li K, Chen S, Fan L, Feng R, Liu H, Liu C, et al. Comparison and evaluation on static application security testing (SAST) tools for java. 2023, p. 921–33. http://dx.doi.org/10.1145/3611643.3616262.
[4] Bennett G, Hall T, Winter E, Counsell S. Semgrep*: Improving the limited performance of static application security testing (SAST) tools. 2024, p. 614–23. http://dx.doi.org/10.1145/3661167.3661262.
[5] Sharma T, Kechagia M, Georgiou S, Tiwari R, Vats I, Moazen H, et al. A survey on machine learning techniques applied to source code. J Syst Softw 2024;209:111934. http://dx.doi.org/10.1016/j.jss.2023.111934.
[6] Casey B, Santos JCS, Perry G. A survey of source code representations for machine learning-based cybersecurity tasks. 2024, http://dx.doi.org/10.48550/arXiv.2403.10646, arXiv preprint arXiv:2403.10646.
[7] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, et al. Automated vulnerability detection in source code using deep representation learning. 2018, p. 757–62. http://dx.doi.org/10.1109/ICMLA.2018.00120.
[8] Wang W, Nguyen TN, Wang S, Li Y, Zhang J, Yadavally A. DeepVD: Toward class-separation features for neural network vulnerability detection. In: 2023 IEEE/ACM 45th international conference on software engineering. IEEE; 2023, p. 2249–61. http://dx.doi.org/10.1109/ICSE48619.2023.00189.
[9] Li H, Hao Y, Zhai Y, Qian Z. The hitchhiker's guide to program analysis: A journey with large language models. 2023, arXiv:2308.00245.
[10] Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: Are we there yet? IEEE Trans Softw Eng 2021;48:3280–96. http://dx.doi.org/10.1109/TSE.2021.3087402.
[11] Sonnekalb T, Heinze T, Mäder P. Deep security analysis of program code: A systematic literature review. Empir Softw Eng 2022;27. http://dx.doi.org/10.1007/s10664-021-10029-x.
[12] Bhuiyan M, Parthasarathy A, Vasilakis N, Pradel M, Staicu C-A. SecBench.js: An executable security benchmark suite for server-side JavaScript. 2023, p. 1059–70. http://dx.doi.org/10.1109/ICSE48619.2023.00096.
[13] Owasp-benchmark. 2016, URL https://github.com/OWASP-Benchmark/BenchmarkJava. (Accessed 27 May 2025).
[14] SAST benchmark. 2024, URL https://github.com/Perdiga/sast-benchmark. (Accessed 27 May 2025).
[15] OpenSSF CVE benchmark. 2020, URL https://github.com/ossf-cve-benchmark/ossf-cve-benchmark. (Accessed 27 May 2025).
[16] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. CodeBERT: A pre-trained model for programming and natural languages. 2020, p. 1536–47. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.
[17] Ferenc R, Hegedüs P, Gyimesi P, Antal G, Bán D, Gyimothy T. Challenging machine learning algorithms in predicting vulnerable JavaScript functions. 2019, p. 8–14. http://dx.doi.org/10.1109/RAISE.2019.00010.
[18] Ferenc R, Viszkok T, Aladics T, Jász J, Hegedűs P. Deep-water framework: The swiss army knife of humans working with machine learning models. SoftwareX 2020;12:100551.
[19] Team Q. Qwen2.5: A party of foundation models. 2024, URL https://qwenlm.github.io/blog/qwen2.5/.
[20] Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, et al. Transformers: State-of-the-art natural language processing. 2020, p. 38–45. http://dx.doi.org/10.18653/v1/2020.emnlp-demos.6.
[21] OpenSSF CVE benchmark for AI. 2025, URL https://github.com/viszkoktamas/ossf-cve-benchmark. (Accessed 27 May 2025).
[22] ossf-sast-ml. 2025, URL https://github.com/viszkoktamas/ossf-sast-ml. (Accessed 27 May 2025).
[23] Zhou X, Tran D-M, Thanh L, Zhang T, Irsan I, Sumarlin J, et al. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. 2024, http://dx.doi.org/10.48550/arXiv.2407.16235.