Type of Contribution:
Research Paper
Review Paper
Case Study

ENERGY: JURNAL ILMIAH ILMU-ILMU TEKNIK
Vol. 15, No. 2 (2025) pp 239-250
DOI: 10.51747/energy.v15i2.15210

E-ISSN: 2962-2565

This article contributes to:

Implementation of Static Code Analysis to Detect Vulnerabilities in Applications Developed with the Assistance of Large-Language Models (LLM)

Arnold Nasir[1]*, Kasmir Syariati[1], Citra Suardi[1], David Sundoro[1], Juan Salao Biantong[1], Reinaldo Lewis Lordianto[1]

[1] Informatics (Makassar City Campus), Ciputra University, 60219, **Indonesia**

*arnold.nasir10@gmail.com

Abstract

The emergence of large language models (LLMs), such as ChatGPT and GitHub Copilot, has transformed software development, including in higher education. Students can now easily create PHP code for Laravel web applications. This research implements static code analysis with PHPStan to detect security vulnerabilities in student-developed PHP code that is likely assisted by LLMs. The analysis was performed on the full code of 28 capstone projects, focusing on student projects that demonstrated patterns consistent with heavy LLM output use. The results show that 64.16% of LLM-assisted code often neglects data sanitization, uses raw queries without parameterization, and contains vulnerable authentication logic. This study contributes to web application security literacy for students and recommends static analysis as a pedagogical and preventive tool.

Keywords: Large Language Models, Code Security, Static Code Analysis, Web Development

Article Info
Submitted: 2025-10-25
Revised: 2025-11-28
Accepted: 2025-11-29
Published: 2025-11-30

Publisher
Universitas Panca Marga

1. Introduction

Advances in machine learning and artificial intelligence over the past decade have fundamentally reshaped software engineering workflows. The emergence of Large Language Models (LLMs), capable of generating code, explaining algorithms, and offering refactoring suggestions, has driven the rise of AI-based code assistants or Code Generation Tools (CGTs). These tools, trained on a large corpus of source code, now function as AI "pair programmers" widely used in professional

development and university-level programming courses [1], [2]. Their ability to translate natural language descriptions into functional code has lowered the barrier to entry for beginners and accelerated the prototyping process in academic settings [3].

Despite these clear benefits, the rapid integration of LLMs into undergraduate coursework raises serious concerns about how students develop programming competence. While early studies emphasized productivity gains and reduced frustration during coding tasks [4], [5] emerging evidence warns that the reliance on LLM-generated solutions may weaken computational thinking and lead to superficial understanding of program behavior [6], [7], [8], [9]. Instead of engaging in iterative reasoning, students often adopt a prompt-driven workflow that shifts programming from constructing solutions to selecting and adapting model outputs [10].

Security concerns amplify these educational issues. Unlike compilers or static analyzers, LLMs generate code probabilistically and do not perform semantic verification or enforce secure design principles [11], [12]. As a result, they frequently produce solutions that are functionally correct at the surface level but inherently insecure in structure. Controlled studies have shown that LLMs can perform competitively on introductory programming assessments, scoring up to 78% on typical exams, but still embed serious vulnerabilities in the generated code[1], [13]. The reliance on these tools also fosters a "false sense of security" [14], leading novices to bypass critical security inspection.

Recent security-focused analyses report that nearly 40% of LLM-generated programs contain at least one exploitable weakness [12]. Common patterns observed include Improper Input Validation (CWE-20), SQL Injection (CWE-89), Broken Access Control (CWE-285, CWE-639), Authentication Failures (CWE-287), and Hardcoded Credentials (CWE-798). These issues align with critical risks identified in the OWASP Top 10 and can compromise confidentiality, integrity, and system availability [15], [16]. The underlying cause stems from the nature of LLMs which they reproduce statistical patterns found in training data including insecure idioms without understanding security invariants [17], [18]. For instance, LLMs often generate raw SQL queries without sanitization or authentication logic, lacking proper authorization checks [19].

Existing literature tends to evaluate correctness, usability, or code quality in isolation, leaving a critical gap in understanding how LLM-generated code impacts novice developers in real academic environments. Few studies examine the intersection of learning behavior, code security, and AI-assisted development, particularly in undergraduate settings where students may treat AI suggestions as authoritative.

This study addresses these gaps by empirically analyzing vulnerabilities in real-world PHP/Laravel applications developed by students with LLM assistance. Through static code analysis using PHPStan and mapping to the OWASP Threat and Safeguard Matrix (TaSM) [23], the research aims to identify prevailing vulnerability patterns, understand how LLM usage influences security risks in educational contexts, and demonstrate the potential of static analysis as an early-stage pedagogical intervention.

2. Methods

This study employs an empirical research method that combines static code analysis and vulnerability profiling to evaluate security weaknesses in Laravel-based PHP applications developed with the assistance of Large Language Models (LLMs). The projects developed by the students have the following configuration:

1. XAMPP ver 8.1.25 (PHP 8.1.25, MariaDB 10.4.32)
2. Windows 11
3. Laravel 11.9

The methodology is designed to enable reproducibility through clear documentation of dataset selection, analysis tools, categorization schemes, and validation steps, as seen in **Figure 1**. The research workflow consists of dataset acquisition, filtering and validation, static code analysis, vulnerability classification, and result interpretation.

2.1 Research Stages

This study adopts a descriptive-analytic design to examine the security posture of student-developed web applications and to investigate patterns of vulnerabilities linked to the implicit use of LLMs in programming tasks. The research design focuses on two primary objectives: (1) to assess the prevalence of security vulnerabilities in student code using static analysis, and (2) to classify and interpret vulnerability patterns based on OWASP risk categories. The study integrates quantitative measures (e.g., frequency of CWE categories) and qualitative inspection (manual verification of code) to strengthen analytical validity.

2.2 Data Collection and Selection

The data for this study consists of capstone-level web application projects developed by undergraduate students enrolled in a Web Development course. A total of 30 projects were initially collected from students' public GitHub repositories. Each project implemented a specific system using the Laravel framework to fulfill instructional learning objectives. Only 28 projects were included in the analysis after applying the following selection criteria, as seen in **Table 1**.

Tabel 2. Project Criteria to be Used in The Research

Inclusion Criteria	Exclusion Criteria
• The project must be a functional web application built using Laravel 11. • The codebase must be publicly accessible via GitHub. • The project must include application logic, routing, database access, and authentication modules.	• Projects not using Laravel 11 (1 eliminated). • Projects that failed to run due to missing dependencies or broken configuration (1 eliminated).

Although the use of LLMs, such as ChatGPT or GitHub Copilot, was neither required nor explicitly documented by students, an initial survey and code pattern inspection were used to identify likely LLM usage. Indicators included repetitive coding patterns, generic error messages, boilerplate CRUD templates, and identical function naming patterns typical of AI-generated suggestions.

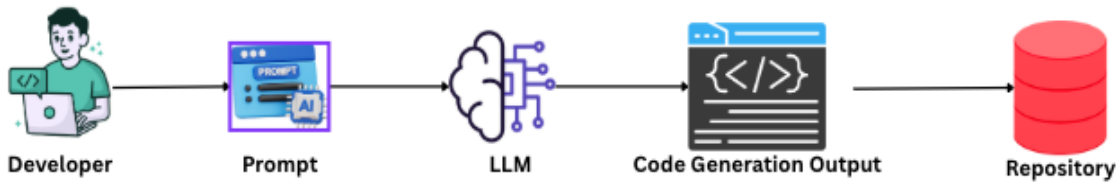

Figure 1. Process of Developing Datasets

2.3 Vulnerability Categories

Vulnerability detection in this study was conducted using the Common Weakness Enumeration (CWE) taxonomy mapped to security risk categories from OWASP [20]. Only code-level weaknesses relevant to PHP and Laravel were analyzed. Vulnerabilities were grouped into five primary categories, as seen in **Table 2**.

These categories were selected based on their prevalence in prior empirical studies on LLM-generated code and their alignment with the OWASP Top 10 (2021) web application security risks.

2.4 Static Analysis Procedure

Static analysis was conducted using PHPStan configured at Level 8, which enforces strict static type and structure validation. Although PHPStan is primarily designed for code consistency and reliability checks, rather than explicit security scanning, its rule engine has been extended with custom configurations to detect

patterns associated with security weaknesses. Thus, static analysis was applied to the 28 validated Laravel projects, specifically targeting the /app, /routes, and /config directories. These extensions targeted insecure Laravel usage patterns, such as:

a) Direct use of raw database queries (DB::select, DB::statement) without query binding
b) Missing input validation using $request->validate()
c) Route definitions without authentication or authorization middleware
d) Unsafe file handling (Storage::get or File::get without sanitization)

Tabel 2. Vulnerability Categories According to CWE Reference

Category	Description	CWE Reference
Improper Input Validation	Lack of sanitization/validation of user inputs	CWE-20
Broken Access Control	Unprotected routes or unauthorized operations	CWE-285, CWE-639
SQL Injection	Unsafe query execution via raw input	CWE-89
Authentication Failures	Weak or missing authentication controls	CWE-287
Hardcoded Secrets	API keys or credentials stored in source code	CWE-798

For each repository, PHPStan was executed recursively on the /app, /routes, and /config directories using a standardized configuration file to maintain consistency across all analyses. The tool generated structured reports containing detected issues, which were then exported in JSON format for post-processing and classification.

2.5 Construction of Vulnerability Matrix

To provide a structured interpretation of detected weaknesses, results were mapped into a Vulnerability Threat Matrix adapted from the OWASP Threat and Safeguard Matrix (TaSM). This matrix not only quantifies the security posture of the analyzed projects but also provides a pedagogical tool for students to understand how vulnerabilities evolve from coding errors to exploitable conditions. Each entry is classified by five dimensions: Threat Source, Weakness Type (CWE), Attack Vector, Impact, and Recommended Mitigation, as seen in **Table 3**.

The final vulnerability matrix was validated through manual inspection by two reviewers with secure coding expertise to ensure consistency and eliminate false positives.

Tabel 3. Vulnerability Matrix based on TaSM

Threat Source	Weakness Type (CWE)	Attack Vector	Potential Impact	Recommended Mitigation
External user / API client	CWE-20	Unsanitized user input via forms or API parameters	Data corruption, injection, or XSS attacks	Implement Laravel validation rules using Request::validate() and sanitize input data.
Unauthorized user/endpoint	CWE-285	Accessing unprotected routes or hidden endpoints	Unauthorized data access, privilege escalation	Use Laravel middleware (auth, can) and define route authorization in web.php.
Malicious query/attacker	CWE-89	Direct query execution using DB::select() with unsanitized variables	Data theft, database corruption	Use Laravel Eloquent ORM or parameterized queries to prevent SQL injection.
Insider/developer oversight	CWE-798	Credentials or API keys stored in source files	Unauthorized system or API access	Store secrets in .env file; never commit credentials to version control.
Unauthenticated user	CWE-287	Missing or weak authentication mechanism	Unauthorized system entry, session hijacking	Implement Laravel authentication scaffolding and hashed passwords using bcrypt.

3.Results and Discussion

3.1 Quantitative Analysis of Vulnerability Patterns

Static analysis was performed on 28 validated Laravel 11 projects using PHPStan at level 8. The analysis identified a total of 173 vulnerability instances distributed across five primary CWE categories. **Tabel 4** presents the detailed distribution of vulnerabilities per category.

The data indicate that Improper Input Validation (CWE-20) and Broken Access Control (CWE-285) account for approximately 64.16% of all detected vulnerabilities. These weaknesses represent fundamental lapses in secure coding awareness rather than complex attack surfaces.

Moreover, projects that relied more heavily on generated snippets showed a 24.27% higher occurrence of CWE-89 and CWE-798 vulnerabilities, suggesting an increased dependency on insecure example patterns embedded in the generated code.

3.2 Example of Vulnerability Code Snippet

The research uncovered several concrete code snippets with identified vulnerabilities, as detailed below:

a. SQL Injection:

```
$query = "SELECT * FROM users WHERE email = '" . \$_POST['email'] ."'";
```

```
$result = DB::select($query);
```

b. Authentication Failures (Login without token validation or rate-limiting):

```
if (\$request->input('password') == 'admin123') {
    a. Auth::loginUsingId(1);
}
```

c. Hardcoded Credentials (Store credentials explicitly):

```
\$apiKey = "sk_test_51H...";
```

d. Improper Input Validation:

```
\$book = new Book();
\$book->title = \$_POST['title'];
\$book->save();
```

e. Broken Access Control:

```
Route::get('/admin/deleteUser', [AdminController::class, 'delete']
```

Tabel 4. Vulnerability Frequency by CWE Category

CWE Category	Occurrences	Percentage (%)	Severity (High/Med/Low)
CWE-20 Improper Input Validation	64	36.99	High
CWE-285 Broken Access Control	47	27.17	High
CWE-89 SQL Injection	23	13.29	High
CWE-798 Hardcoded Credentials	19	10.98	Medium
CWE-287 Authentication Failures	20	11.57	Medium
Total	173	100	—

3.3 Root Cause of Security Weaknesses

The vulnerabilities identified in LLM-assisted code are not isolated occurrences but the result of deeper behavioral, cognitive, and contextual shortcomings in how novice developers interact with AI-based code generation tools. The root causes discussed below were not derived from primary data (e.g., interviews or surveys), but were systematically inferred through a qualitative, manual review of the 173 vulnerable code segments identified by PHPStan's static analysis, cross-referenced with established educational and security literature.

a. Over-Trust in Generated Code (False Sense of Correctness)

One of the most pervasive causes of security weaknesses is the over-trust novice developers place in the syntactic and functional correctness of generated code. Students often assume that since the code compiles and runs without errors, it must also be secure and efficient. This false sense of security discourages deeper inspection and rigorous security testing, aligning directly with the findings of Perry et al. [14], which documents that the use of AI assistants increases the likelihood of insecure code introduction among developers. In several analyzed projects, students directly deployed code suggested by the model without manual verification, resulting in exploitable flaws such as unsanitized SQL queries and missing authentication logic.

For example, one project included the following vulnerable snippet:

```
$results = DB::select("SELECT * FROM users WHERE email = '$request->email'");
```

The above code appears functional and produces correct query results, yet it allows for SQL injection due to unsanitized user input. The student relied entirely on model output, assuming correctness because the query executed successfully.

b. Lack of Contextual Validation (Input Handling and Access Control)

LLM-generated code typically lacks context awareness beyond the immediate prompt, resulting in incomplete implementations that omit critical validation or access control mechanisms. This limitation is particularly evident in frameworks like Laravel, where developers must explicitly define middleware or input validation layers.

In multiple cases, generated controller methods failed to use built-in validation features such as Request::validate() or form request classes. Similarly, routes lacked middleware enforcement, allowing unrestricted access to administrative endpoints. These omissions stem from the model's inability to infer application-level security policies and the student's inexperience in compensating for such gaps.

```
Route::get('/admin', [AdminController::class, 'index']);
```

Without explicit authentication middleware, the above route exposes sensitive functionality to unauthorized users. Students often overlooked this omission because the page rendered correctly during testing, reinforcing their misplaced trust in the generated code. This behavior is indicative of cognitive offloading[21], where students prioritize functional completion over a deeper understanding of security invariants.

c. Copy-Paste and Prompt Reuse Behavior

Another observed contributor is the copy-paste culture reinforced by prompt reuse behavior. Many students reused generated snippets from prior tasks or online sources without fully understanding their functionality. This led to inconsistencies in data validation and security policies across the same application.

For instance, a student reused an earlier API controller template generated by the model, resulting in mismatched input sanitization between modules. Inconsistent application of security controls made the system more susceptible to injection and logic manipulation attacks. The convenience of LLM outputs inadvertently encouraged fragmented and redundant coding practices rather than deliberate, secure design [22].

3.4 Security Implications and Industry Practices

Based on these research findings, it can be concluded that using LLM without a basic understanding of security poses a high risk. Therefore, for industrial contexts, the adoption of static code analysis tools must be an integral part of the DevSecOps workflow. This research can also provide a real contribution to the development of secure guidelines for the use of LLMs by novice developers. If these findings are integrated into industrial and academic practices, the productivity generated by LLM can be in line with the security of the resulting system.

4. Conclusion

This research demonstrates that the use of LLMs in application development by students, while accelerating the programming process, still has implications for the security aspects of the resulting software. Analysis of 28 LLM-based Laravel projects revealed a uniform pattern of error, dominated by fundamental weaknesses in Improper Input Validation (CWE-20) and Broken Access Control (CWE-285), which together constitute over 64% of all 173 detected instances. This uniformity indicates a critical over-reliance on LLM output without adequate human security review, fostering a false sense of security among students.

The study's primary contribution is twofold: it provides empirical evidence of specific security risks in LLM-assisted PHP/Laravel code and demonstrates the efficacy of a PHPStan-based static analysis pipeline, enhanced with custom rules, as a scalable method for early vulnerability detection.

From a practical and pedagogical standpoint, these findings underscore the necessity of integrating security literacy directly into computer science curricula. Academic institutions must emphasize the systematic use of automated tools as a preventive measure and utilize frameworks like the OWASP Threat and Safeguard

Matrix (TaSM) to translate abstract security concepts into concrete mitigation actions.

For future research, we recommend expanding the analysis by integrating multi-tool static analysis to cover more complex logical flaws and conducting an empirical experiment using a TaSM-based pedagogical intervention to measure its effectiveness in improving students' secure coding proficiency when they utilize LLMs.

Authors' Declaration

Authors' contributions and responsibilities - The authors made substantial contributions to the conception and design of the study. The authors took responsibility for data analysis, interpretation, and discussion of results. The authors read and approved the final manuscript.

Funding - Basic Research Program for Beginners (PDP) from the Directorate of Research, Technology, and Community Service (DPPM) of the Ministry of Higher Education, Science, and Technology.

Availability of data and materials - All data is available from the authors.

Competing interests - The authors declare no competing interest.

Additional information - No additional information from the authors.

References

[1] H. Tian et al., "Is ChatGPT the Ultimate Programming Assistant -- How far is it?," Aug. 31, 2023, arXiv: arXiv:2304.11938. doi: 10.48550/arXiv.2304.11938.

[2] S. Lau and P. Guo, "From 'Ban It Till We Understand It' to 'Resistance is Futile': How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot," in Proceedings of the 2023 ACM Conference on International Computing Education Research V.1, Chicago IL USA: ACM, Aug. 2023, pp. 106–121. doi: 10.1145/3568813.3600138.

[3] J. Savelka, A. Agarwal, C. Bogart, Y. Song, and M. Sakr, "Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?," in Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, June 2023, pp. 117–123. doi: 10.1145/3587102.3588792.

[4] I. R. da S. Simões and E. Venson, "Evaluating Source Code Quality with Large Language Models: a comparative study," Sept. 22, 2024, arXiv: arXiv:2408.07082. doi: 10.48550/arXiv.2408.07082.

[5] A. Hellas, J. Leinonen, S. Sarsa, C. Koutcheme, L. Kujanpää, and J. Sorva, "Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests," in Proceedings of the 2023 ACM Conference on International Computing Education Research V.1, Chicago IL USA: ACM, Aug. 2023, pp. 93–105. doi: 10.1145/3568813.3600139.

[6] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," Apr. 13, 2024, arXiv: arXiv:2308.04838. doi: 10.48550/arXiv.2308.04838.

[7] J. Savelka, A. Agarwal, M. An, C. Bogart, and M. Sakr, "Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses," in Proceedings of the 2023 ACM Conference on International Computing Education Research V.1, Chicago IL USA: ACM, Aug. 2023, pp. 78–92. doi: 10.1145/3568813.3600142.

[8] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?," Jan. 06, 2024, arXiv: arXiv:2204.04741. doi: 10.48550/arXiv.2204.04741.

[9] P. Denny et al., "Computing Education in the Era of Generative AI," Commun. ACM, vol. 67, no. 2, pp. 56–67, Feb. 2024, doi: 10.1145/3624720.

[10] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming Is Hard -- Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation," Dec. 02, 2022, arXiv: arXiv:2212.01020. doi: 10.48550/arXiv.2212.01020.

[11] S. Dou et al., "What's Wrong with Your Code Generated by Large Language Models? An Extensive Study," July 08, 2024, arXiv: arXiv:2407.06153. doi: 10.48550/arXiv.2407.06153.

[12] A. M. Dakhel et al., "GitHub Copilot AI pair programmer: Asset or Liability?," Apr. 14, 2023, arXiv: arXiv:2206.15331. doi: 10.48550/arXiv.2206.15331.

[13] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "ChatGPT and Software Testing Education: Promises & Perils," in 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Apr. 2023, pp. 4130–4137. doi: 10.1109/ICSTW58534.2023.00078.

[14] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen Denmark: ACM, Nov. 2023, pp. 2785–2799. doi: 10.1145/3576915.3623157.

[15] "OWASP Top Ten | OWASP Foundation." Accessed: Oct. 24, 2025. [Online]. Available: https://owasp.org/www-project-top-ten/

[16] S. Elder, N. Zahan, V. Kozarev, R. Shu, T. Menzies, and L. Williams, "Structuring a Comprehensive Software Security Course Around the OWASP Application Security Verification Standard," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), Madrid, ES: IEEE, May 2021, pp. 95–104. doi: 10.1109/ICSE-SEET52601.2021.00019.

[17] M. L. Siddiq, J. C. S. Santos, S. Devareddy, and A. Muller, "SALLM: Security Assessment of Generated Code," in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, Oct. 2024, pp. 54–65. doi: 10.1145/3691621.3694934.

[18] J. J. Wu, "Large Language Models Should Ask Clarifying Questions to Increase Confidence in Generated Code," Jan. 22, 2024, arXiv: arXiv:2308.13507. doi:

10.48550/arXiv.2308.13507.

[19] C. Zhang, Z. Wang, R. Mangal, M. Fredrikson, L. Jia, and C. Pasareanu, "Transfer Attacks and Defenses for Large Language Models on Coding Tasks," Nov. 22, 2023, arXiv: arXiv:2311.13445. doi: 10.48550/arXiv.2311.13445.

[20] "CWE - Common Weakness Enumeration." Accessed: Oct. 24, 2025. [Online]. Available: https://cwe.mitre.org/

[21] R. Zviel-Girshin, "The Good and Bad of AI Tools in Novice Programming Education," Educ. Sci., vol. 14, no. 10, p. 1089, Oct. 2024, doi: 10.3390/educsci14101089.

[22] G. Fan, D. Liu, R. Zhang, and L. Pan, "The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches," Int. J. STEM Educ., vol. 12, no. 1, p. 16, Mar. 2025, doi: 10.1186/s40594-025-00537-3.