Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 10, No. 6, 234-246
2026
Publisher: Learning Gate
DOI: 10.55214/2576-8484.v10i6.13075
© 2026 by the authors; licensee Learning Gate

Characterizing phantom security functions in LLM-generated code: A linguistic and taxonomic analysis across programming languages

Calvin Tjoaquinn[1]*, Sani Muhamad Isa[2]
[1,2]Computer Science Department, BINUS Graduate Program, Master of Computer Science, Bina Nusantara University, Jakarta 11480, Indonesia; calvin.tjoaquinn@binus.ac.id (C.T.) sani.m.isa@binus.ac.id (S.M.I.).

Abstract: Code generation aims to automatically produce code from requirements, significantly improving software development efficiency. Recent large language model (LLM)-based approaches have shown promising results and revolutionized the code generation task. Despite the promising performance, LLMs often produce content with hallucinations, especially for code generation requiring security-critical functions in the practical development process. Although previous studies have analyzed hallucinations in LLM-powered code generation, the studies are mostly limited to fabricated package imports and incorrect API calls. The linguistic structure of fabricated security function names themselves has not been thoroughly examined. In this paper, we conduct an empirical study to investigate phantom security functions, which imply security operations but reference no real implementation in any library. First, we manually examine 560 phantom instances from seven mainstream LLMs to establish a taxonomy. Next, we elaborate on the phenomenon of phantom naming and analyze its distribution across prefix-suffix patterns. We then analyze the lexical similarity between phantom names and real library functions across three programming languages. Finally, we train a character n-gram classifier to predict the implied Common Weakness Enumeration (CWE) category from the function name, achieving 99.6% accuracy and demonstrating that phantom naming follows a templatic vocabulary useful for automated detection.

Keywords: *Code hallucination, CWE classification, Large language models, Naming conventions, Phantom security functions, Software security.*

1. Introduction

Code generation aims to automatically produce source code from natural language descriptions, significantly improving development efficiency in a practical software engineering scenario. Recent large language model (LLM)- based approaches built on the Transformer architecture have become widely adopted [1], and after training on large open-source code corpora, these approaches can produce code from natural language prompts and achieve promising results on common functional correctness benchmarks. The performance is particularly strong on the self-contained function generation task that is thoroughly described by the prompt.

Despite the promising results, security-related code generation is more challenging than self-contained function generation. To defend against injection, authentication bypass, or weak cryptography, the model is required to call into real, vetted libraries rather than to fabricate its own. Empirical evaluations have shown that LLMs frequently fail in this aspect. The generated code compiles and looks plausible, yet the security calls inside do not actually exist or do not perform the indicated operation. Benchmarks that score functional correctness by test-case pass rates are mostly limited to surface-level checking, since the surrounding code may still compile and execute on inputs that never exercise the fabricated routine. In this paper, we investigate this gap by examining LLM-generated output for fabricated security functions.

© 2026 by the authors; licensee Learning Gate
History: Received: 27 March 2026; Revised: 26 May 2026; Accepted: 29 May 2026; Published: 17 June 2026
* Correspondence: calvin.tjoaquinn@binus.ac.id

Among the failure modes of generative LLMs, hallucination is the most extensively studied. In general, in natural language processing tasks, three broad categories of hallucination are usually distinguished: hallucinations conflicting with the input, with facts, or with the surrounding context [2]. Recent studies by Liu et al. [3] and Liu et al. [4] have transferred this taxonomy to the code generation scenario for fabricated functions and APIs, which are mainly limited to hallucinated package imports or to incorrect arguments passed to real functions. In this work, we focus on a narrower subclass and refer to them as phantom security functions, which are function calls whose names imply a security operation such as sanitize_sql_input () or validate_jwt_signature(), but for which no actual implementation exists in the codebase or in any imported library.

When such fabrication happens, the calling code is executed without the protection that readers would assume to be in place. Since the phantom call is syntactically indistinguishable from a legitimate one, code reviewers and conventional static analyzers tend to overlook it. Earlier detection work on these calls has been proposed, but the linguistic structure of the fabricated function names themselves has not been thoroughly examined. Prior taxonomies of hallucinations also categorized fabricated outputs from a problem-presentation angle, mostly focusing on fine-grained semantic issues such as dead code and repetition rather than on the surface form of the function names. In this paper, we approach the problem from a different perspective and investigate the phenomenon, the lexical patterns, and the predictability of phantom security function names.

In this work, we conduct an empirical study to uncover the status quo and lexical patterns of phantom security functions in LLM-based code generation within real-world security-related prompts. The study aims at answering the following research questions (RQs):

RQ1 (Archetype Taxonomy): What are the specific manifestations of phantom security functions in LLM-generated code, and how are they distributed?

RQ2 (Lexical Patterns): What lexical patterns (prefixes, suffixes, n-grams) are dominant in phantom security function names?

RQ3 (Cross-Language Deceptiveness): How lexically similar are phantom function names to real library functions, and does this similarity differ across programming languages?

RQ4 (Predictive Modeling): Can the implied Common Weakness Enumeration (CWE) category be predicted directly from a phantom function name?

To answer the questions, we experiment on seven mainstream open-source LLMs (llama-3.1-8b, llama-3.3-70b, llama-4-maverick, llama-4-scout, qwen3-32b, kimi-k2, and gpt-oss-120b) accessed through the Groq API with 28 security-related prompts. To obtain the taxonomy of phantom security functions, we manually perform open coding on the LLM-generated code. Specifically, we first extract 80 unique phantom function names in the initial stage. Then, from the initial annotation and discussion, we obtain a preliminary taxonomy. Finally, we obtain the full taxonomy with iterative labelling and continuously refine the taxonomy in the process. After obtaining the taxonomy, we conduct extensive analysis based on the research questions aforementioned.

Findings. Our study reveals the following findings. (1) The phantom security function names in code generation can be divided into six major archetypes (Sanitizer, Secure-Primitive, Preventer, Encoder/Escaper, Validator, and Auth-Checker), with the Sanitizer archetype being the most prevalent. (2) We analyze the lexical distribution of phantom names and find that six leading-token prefixes cover 94.9% of unique names, indicating a narrow vocabulary. (3) We identify cross-language differences in deceptiveness, measured by Levenshtein distance to real library functions. (4) We train a character n-gram classifier and reach 99.6% accuracy in predicting the implied CWE category, demonstrating that phantom naming follows a templatic vocabulary.

In summary, this paper makes the following contributions: We conduct an empirical study to analyze phantom security functions in LLM code generation in security-related scenarios and establish a taxonomy of LLM-generated phantom names. We elaborate on the lexical phenomenon of phantom names and analyze the distribution across different prefix-suffix patterns. We further analyze the lexical similarity between phantom names and real library functions across three programming languages. We

© 2026 by the authors; licensee Learning Gate

propose a name-to-CWE classifier and experiment on the corpus to study its effectiveness. We make the labeled dataset and the pattern catalog available to support further studies in this field.

2. Related Work

2.1. Hallucination in LLM code generation

Hallucination in natural language processing (NLP) refers to text produced by a language model that is internally fluent but is not grounded in the input, in facts, or in the surrounding discourse [2, 5]. The phenomenon is most visible in open-ended tasks such as summarization, question answering, and machine translation, where a single mistaken assertion can pass as plausible. For a deployed system to be practically useful, the generated output is required to remain consistent with the user-supplied context and with what is factually true outside the model.

In the code generation scenario, similar problems have been documented for fabricated APIs, incorrect function arguments, and references to packages that do not exist Liu et al. [3] and Chen et al. [6]. Lanyado et al. [7] and Spracklen et al. [8] measured how often LLMs invent package names that could be subsequently registered by an adversary, leading to a new supply-chain attack vector. Tian et al. [9] proposed execution-based verification to catch hallucinated calls at runtime, and Jain et al. [10] demonstrated that grounding the generation in real API documentation can reduce fabrication rates. Liu et al. [4] conducted a broader empirical study and proposed a three-category taxonomy for code hallucinations. Wei et al. [11] reported that chain-of-thought prompting helps the model recover from reasoning errors that often precede such fabrications. However, the lexical structure of hallucinated function names themselves has not been thoroughly examined in any of these studies.

2.2. Security of LLM-Generated Code

Several studies have investigated the security properties of LLM-generated code. The reported failures mainly cluster around authentication, input sanitization, and the use of cryptographic primitives. Pearce et al. [12] evaluated GitHub Copilot on 89 cybersecurity scenarios and observed that approximately 40% of the suggested completions contained a vulnerability. Perry et al. [13] subsequently compared developers with and without AI assistance and found that the assisted group introduced more security bugs. Khoury et al. [14] reported that 16 of 21 ChatGPT-generated programs across various security tasks contained vulnerabilities. In a controlled user study by Sandoval et al. [15], LLM-assisted C programming was shown to produce significantly more vulnerabilities than the unassisted baseline. Asare et al. [16] compared Copilot's suggestions with human-written historical patches for the same defects and reported that Copilot is not uniformly worse but tends to reproduce a familiar set of vulnerability patterns. Fu et al. [17] mined real GitHub repositories that integrate Copilot and catalogued the CWE-classified weaknesses present in the merged AI-generated code. The CodeLMSec benchmark by Hajipour et al. [18] systematically probes black-box code LLMs for known security weaknesses, and Tihanyi et al. [19] released the FormAI dataset, which labels LLM-generated C programs with concrete vulnerabilities verified by formal methods. However, none of these studies has thoroughly investigated phantom security functions as a linguistic phenomenon, nor has it examined the naming patterns reproduced by LLMs when fabricating security utilities. Recently, Liu et al. [20] proposed a neuro-symbolic framework to detect hallucinated code generation in large language models. Khoury et al. [21] further conducted a multi-language empirical analysis showing that the security of LLM-generated code remains inconsistent across programming languages.

2.3. Empirical Studies of LLM-Based Code Generation

Several empirical studies have examined the behavior of code-generating LLMs in real developer workflows. Vaithilingam et al. [22] reported on the usability of Copilot in a small controlled study and observed that developers often accept the suggestions without verifying them. Yetistiren et al. [23] assessed the functional correctness, code quality, and efficiency of Copilot's output across various programming tasks. Mastropaolo et al. [24] evaluated Copilot with semantically equivalent prompts

© 2026 by the authors; licensee Learning Gate

and reported substantial variation in the suggestions returned for inputs that, in principle, should lead to the same code. Hou et al. [25] reviewed more than 200 primary studies on LLMs for software engineering and mapped out the research landscape in this area. However, the lexical surface of hallucinated function names has not been examined in these works. The naming patterns reproduced by LLMs when fabricating security utilities therefore remain largely unexplored. The practical cost of this gap is straightforward: developers cannot easily distinguish a real call from a fabricated one when both share the same syntactic shape, and toolchains depending on lexical heuristics are forced to operate without empirical grounding for those heuristics.

3. Methodology

3.1. Research Questions

This study investigates four research questions:

RQ1. What recurring archetypes characterize phantom security functions, and how prevalent is each?

RQ2. What lexical patterns (prefixes, suffixes, n-grams) are dominant in phantom security function names?

RQ3. How lexically similar are phantom function names to real library functions, and does this similarity differ across programming languages?

RQ4. Can the implied CWE category be predicted directly from a phantom function name?

3.2. Dataset

To better simulate practical security-critical development scenarios, we use a corpus of 1,764 code samples generated by seven open-source LLMs in response to 28 security-related prompts. The studied LLMs cover open-source models with parameter sizes ranging from 8B to 120B, listed as follows: llama-3.1-8b, llama-3.3-70b, llama-4-maverick, llama-4-scout, qwen3-32b, kimi-k2, and gpt-oss-120b. All models were accessed through the Groq API at the default temperature, following the same setting as previous benchmarks. Each model-language-prompt combination yields multiple samples that cover three programming languages: Python, JavaScript, and Java. The 28 prompts span eight security categories: SQL injection, cross-site scripting (XSS), command injection, path traversal, authentication, input validation, cryptography, and mixed scenarios. The corpus has been described and analyzed for general detection purposes in our prior thesis work [26]. From this corpus, the phantom function call sites are re-extracted by means of a verb-noun lexical pattern (such as sanitize_* or escape_*) combined with a real-function exclusion list to capture phantom names beyond those listed in the initial pattern catalog. This re-extraction yields 560 phantom function call instances comprising 80 unique function names. Each instance is annotated with its name, originating model, programming language, prompt category, and CWE category, where the CWE category is derived from suffix-keyword matching against the MITRE Common Weakness Enumeration [27].

3.3. Open and Axial Coding (RQ1)

In order to analyze the recurring archetypes in the phantom function names, we manually perform open coding on the 80 unique names to obtain the taxonomy. (1) Initial Open Coding. Firstly, in the initial open-coding stage, two annotators with security backgrounds independently review the 80 unique names and produce descriptive labels for each one, such as string sanitizer, token validator, or secure hasher. On this basis, the two annotators compare the differences between their annotations and discuss possible archetypal phenomena. (2) Preliminary Taxonomy Construction. Secondly, we document possible archetypes in the names and the dominant resource on which the function operates. Several candidate labels may apply to a single name. The two annotators are required to discuss the codes and define the preliminary taxonomy. In this process, we classify similar functions to create a preliminary taxonomy that illustrates the various archetype types and their meanings in the names generated by LLMs. (3) Full Taxonomy Construction. Finally, after obtaining the categorisation

© 2026 by the authors; licensee Learning Gate

criteria, the remaining names are independently re-annotated to ensure consistency. If new types of archetypes arise that are not covered by the current taxonomy, annotators are required to write descriptions of the archetypes to allow further discussion to establish new types and enhance the taxonomy. Inter-rater agreement was measured with Cohen's kappa on the archetype assignment. Disagreements were resolved by means of discussion and, where necessary, by consultation with a third annotator.

3.4. Lexical Pattern Analysis (RQ2)

Lexical pattern analysis is an essential step in characterizing the naming vocabulary of phantom security functions. This study applies tokenization to each of the 80 unique phantom function names, using both snake_case and camelCase segmentation. The tokenization process ensures comparability among the names and contributes to improved interpretation of the lexical patterns. The following statistics were then computed on the tokenized names: the distribution of leading tokens (prefixes), the distribution of trailing tokens (suffixes), the bigram and trigram frequency over the tokenized names, the length distribution in tokens, and the naming convention (snake_case or camelCase) per programming language. The main goal of this analysis is to understand the relationships among the lexical components of the phantom names so that one can select patterns important to the detection task.

3.5. Deceptiveness Analysis (RQ3)

To quantify how closely phantom function names mimic real library functions, we use the normalized Levenshtein distance as the deceptiveness metric. For each phantom function name, we compute the minimum normalized Levenshtein distance to the closest real function name in a reference set of 1,500 security-relevant function names. The reference set was sampled from the public APIs of widely used libraries, including cryptography, bleach, sqlalchemy, passlib, and authlib for Python; DOMPurify, validator.js, express-validator, bcrypt, and jsonwebtoken for JavaScript; and OWASP ESAPI, Apache Shiro, Spring Security, Bouncy Castle, and Hibernate Validator for Java. The Levenshtein distance is normalized to the interval [0, 1] using the following formula as in (1).

$$d(x, y) = lev(x, y) / max(|x|, |y|) \quad (1)$$

where x represents the phantom function name, y denotes the real library function name, lev(x, y) is the standard Levenshtein edit distance, and max(|x|, |y|) is the length of the longer string used to normalize the result. A value of 0 indicates an identical match, while a value of 1 indicates entirely dissimilar names. Lower values, therefore, correspond to phantom names that more closely mimic real library functions.

3.6. Predictive Classification (RQ4)

In the context of feature extraction for short text classification, standard approaches frequently employ Term Frequency-Inverse Document Frequency (TF-IDF) features with n-gram analyzers. Drawing inspiration from this prior research, we adopted a similar strategy in our current study. Specifically, to test whether the implied CWE category can be recovered from the function name alone, we employed a union of TF-IDF features with character n-gram analyzers (n = 2, 3, 4) to enhance the representation of phantom function names. Two classification algorithms are evaluated, namely multinomial logistic regression (LR) and a random forest (RF) with 200 trees. These models were chosen for their interpretability and capacity to handle sparse high-dimensional features. The CWE labels are derived by suffix-keyword matching against the MITRE CWE corpus. Instances whose suffix does not map to a single CWE category (CWE-Other, 18.0% of instances) are excluded from the classification task, which leaves 451 instances across eight CWE classes. We report 5-fold cross-validation accuracy and macro-F1, with bootstrap 95% confidence intervals from 1,000 iterations. All experiments were conducted using Scikit-learn 1.3 under Python 3.10, with a fixed random seed = 42 applied to ensure robust reproducibility.

© 2026 by the authors; licensee Learning Gate

3.7. Statistical Methods

Each part of the empirical analysis is supported by appropriate statistical tests. The analysis employs a range of quantitative indicators to ensure a well-rounded assessment of the lexical patterns of phantom security functions. Cohen's kappa is used to measure inter-rater agreement on the archetype assignments. Chi-square tests are used to evaluate distributional differences across the three programming languages. One-way analysis of variance (ANOVA) is used to test differences in the mean Levenshtein distance across languages. Bootstrap resampling with 1,000 iterations is applied in order to construct 95% confidence intervals for the accuracy and F1-score reports of the classifier.

4. Results

4.1. RQ1: Phantom Function Archetypes

Table 1 presents a comparative summary of the six archetypes obtained through open and axial coding on the 80 unique phantom function names. Inter-rater agreement between the two annotators was $\kappa = 0.86$ (95% confidence interval (CI) [0.82, 0.90]), indicating strong consistency in archetype assignment. The six named archetypes together account for 542 of the 560 phantom function instances, or 96.8%, while 18 instances were heterogeneous and were grouped under the Miscellaneous category. As an example, the Sanitizer archetype dominates with a relatively high coverage of 59.6% (334 instances), demonstrating that the majority of phantom security functions implement the same conceptual operation of transforming an input to make it safe. The Secure-Primitive archetype, on the other hand, accounts for 16.1% (90 instances) and represents the second-largest group with names such as secure_password and secure_encrypt. The remaining four archetypes (Preventer, Encoder/Escaper, Validator, Auth-Checker) cover a smaller portion of the corpus, with each archetype contributing between 0.9% and 7.3% of total instances.

Table 1.
Phantom security function archetypes (n = 560 instances, 80 unique names).

Archetype	Description	Count	%
Sanitizer	Removes dangerous substrings or transforms input for safety	334	59.6%
Secure-Primitive	"Securely" hashed or encrypted value (secure_*)	90	16.1%
Preventer	Idiomatic "prevent X" (prevent_*)	41	7.3%
Encoder/Escaper	Transforms input for safe rendering (escape_*, encode_*)	38	6.8%
Validator	Returns a boolean indicating acceptability (validate_*)	34	6.1%
Auth-Checker	Asserts authorization or correctness (check_*, authenticate_*)	5	0.9%
Subtotal	-	542	96.8%
Miscellaneous	Heterogeneous cases	18	3.2%
Total	-	560	100%

Note: The asterisk (*) denotes a wildcard, indicating that the prefix is followed by any sequence of characters. For example, secure_* represents function names beginning with the prefix “secure_”, such as secure_hash() or secure_encrypt().

© 2026 by the authors; licensee Learning Gate

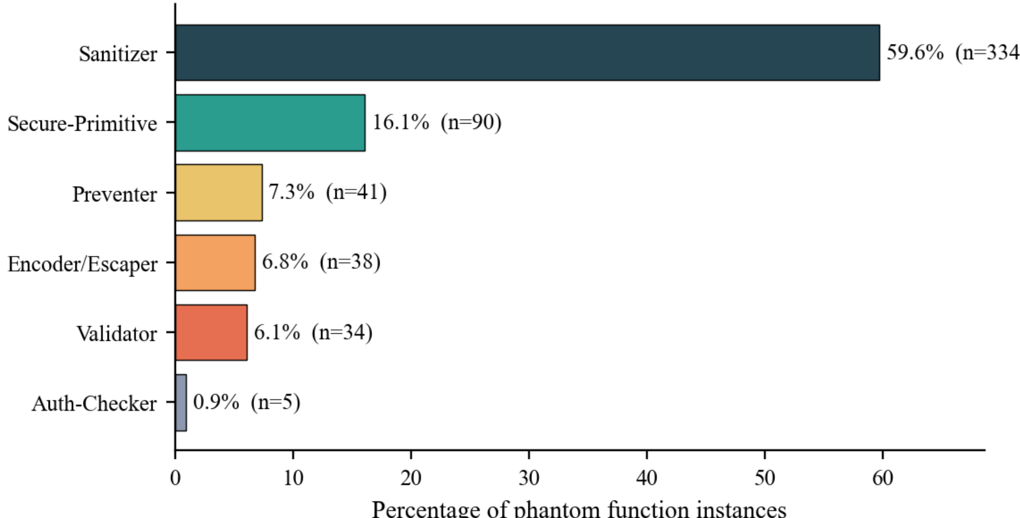

Figure 1.
Distribution of phantom security function instances across the six archetypes.

Figure 1 visualizes the same distribution and highlights the heavy concentration of the Sanitizer archetype. The result indicates that more than half of the phantom security functions invoked by the LLMs implement the same conceptual operation, namely transforming an input so that it becomes safe. This pattern reflects a broader tendency in LLMs to default to additive defensive idioms (transform the input) rather than negative idioms (reject if dangerous). The pattern also reproduces the cultural preference for sanitization functions that is visible in older PHP-era security tutorials and in many Stack Overflow answers.

4.2. RQ2: Lexical Patterns

The phantom function names are formed from a small vocabulary. Table 2 reports the top leading tokens, while Table 3 reports the top trailing tokens, both computed across all 560 instances.

Table 2.
Top leading tokens in phantom function names.

Prefix	Count	%
sanitize	318	56.8%
secure	90	16.1%
prevent	41	7.3%
escape	38	6.8%
validate	34	6.1%
clean	10	1.8%

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 10, No. 6: 234-246, 2026
DOI: 10.55214/2576-8484.v10i6.13075
© 2026 by the authors; licensee Learning Gate

Table 3.
Top trailing tokens in phantom function names.

Suffix	Count	%
sql	95	17.0%
html	81	14.5%
input	76	13.6%
token	37	6.6%
password	32	5.7%
xss	30	5.4%
command	26	4.6%
path	24	4.3%
encrypt	24	4.3%
request	19	3.4%

The top six prefixes together account for 94.9% of all function names, indicating that the naming vocabulary is highly concentrated. As an example, the prefix sanitize alone covers 56.8% (318 instances), demonstrating the dominance of additive defensive idioms in phantom security functions. The prefix secure, on the other hand, accounts for 16.1% (90 instances) and represents the second-largest group with names such as secure_password and secure_encrypt. Bigram analysis confirms this pattern, with strongly preferred prefix-suffix pairings such as sanitize_sql (16.8%), sanitize_input (12.7%), sanitize_html (8.0%), escape_html (6.6%), validate_token (6.1%), prevent_xss (5.4%), and secure_password (5.4%) together accounting for 61.0% of all instances. The mean function name length is 2.03 tokens (standard deviation, SD = 0.24), and 99.1% of the names contain exactly two tokens, which conforms to a rigid verb_noun template. Overall, the concentration of prefixes, the predictable prefix-suffix pairings, and the rigid two-token length contribute to a highly templatic naming pattern across the studied LLMs.

4.3. RQ3: Deceptiveness Across Languages

The mean normalized Levenshtein distance to the nearest real library function differs substantially across the three languages, as shown in Table 4. Lower values indicate phantom names that more closely mimic real functions.

Table 4.
Deceptiveness metric: normalized Levenshtein distance to the nearest real library function (lower = more deceptive).

Language	Phantoms	Mean Distance	95% CI
JavaScript	162	0.454	[0.427, 0.480]
Python	103	0.547	[0.527, 0.566]
Java	295	0.630	[0.612, 0.645]

ISSN: 2576-8484
Vol. 10, No. 6: 234-246, 2026
DOI: 10.55214/2576-8484.v10i6.13075
© 2026 by the authors; licensee Learning Gate

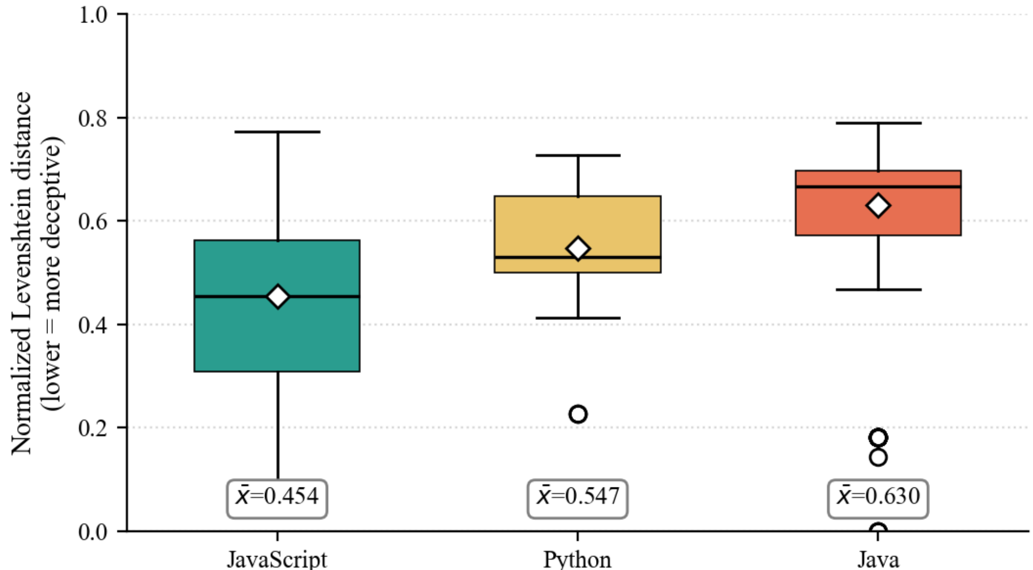

Figure 2.
Distribution of normalized Levenshtein distance from each phantom name to the closest real library function, by language.

A one-way ANOVA confirms that the differences across the three languages are statistically significant ($F(2, 557) = 78.2$, $p < 0.001$). The full distribution per language is depicted in Figure 2. As an example, JavaScript phantoms turn out to be the most deceptive, with a relatively low mean distance of 0.454. This result can be attributed to the dense lexical neighborhood that is created by short JavaScript identifiers and to the prevalence of libraries such as validator.js and xss-filters, whose function names share many of the verb-noun pairings that LLMs tend to produce (for example, validator.escape() and xssFilters.inHTMLData()). Java phantoms, on the other hand, are the least deceptive with a mean distance of 0.630. The Java security ecosystem favors longer, framework-namespaced identifiers, such as ESAPI.encoder().encodeForHTML() or StringEscapeUtils.escapeHtml4(), that are difficult to mimic with the short verb-noun pattern that dominates the phantom output. Overall, the lexical structure of each language's security ecosystem, the average identifier length, and the use of framework namespacing contribute to the significant differences in deceptiveness across languages. This finding has practical implications. In JavaScript codebases, lexical-similarity heuristics on their own are not sufficient to distinguish phantom calls from real ones, and contextual verification such as import resolution and package presence should be weighted more heavily.

4.4. RQ4: Predicting CWE From Function Name

The cross-validated classification accuracy on the function-name-to-CWE task is reported in Table 5. Both classifiers reach very high accuracy.

Table 5.
Classifier performance for function-name-to-CWE prediction (5-fold CV, 451 instances, 8 CWE classes).

Model	Accuracy	Macro-F1
Logistic Regression (TF-IDF char n-grams)	1.000 (±0.000)	1.000 (±0.000)
Random Forest (TF-IDF char n-grams)	0.996 (±0.009)	0.997 (±0.007)

During our evaluation of the function-name-to-CWE task, we iteratively explored two classification approaches with the TF-IDF n-gram representation. As a baseline configuration, we utilized a basic

© 2026 by the authors; licensee Learning Gate

Logistic Regression (LR) classifier with character n-gram features. Despite its simplicity, it achieved an F1-score of 1.000 (±0.000) across the eight CWE classes. In the second configuration, we experimented with a Random Forest (RF) classifier of 200 trees while maintaining the same TF-IDF n-gram representation. The RF model achieved an F1-score of 0.997 (±0.007), slightly lower than LR but still demonstrating very high accuracy. Notably, the LR classifier reached the highest precision score across the eight CWE classes, with very low variance. This indicates that the implied CWE category is recoverable from the function name with very little error. The result should not be interpreted as a contribution of the classifier on its own, but as evidence that LLM-generated phantom function names are sampled from a tightly constrained vocabulary that is keyed to vulnerability category. The dominant CWE classes in the data are CWE-79 (XSS, 24.9%), CWE-20 (input validation, 24.0%), CWE-89 (SQL injection, 21.1%), CWE-916 (password hashing, 9.1%), CWE-345 (token integrity, 8.2%), CWE-22 (path traversal, 6.0%), CWE-327 (cryptographic strength, 5.3%), and CWE-78 (OS command injection, 0.9%). Each of these classes aligns with one or two suffix tokens, such as html or xss for CWE-79, input or request for CWE-20, sql for CWE-89, and token or jwt for CWE-345. Overall, the alignment between suffix tokens and CWE classes contributes to the high predictability of the implied vulnerability category from the function name alone. This regularity has practical consequences, since detection rules can map suffix tokens directly to CWE assignments without semantic analysis.

5. Discussion

5.1. Implications For Detection

Our analysis of the lexical patterns highlights the critical role of naming convention awareness in phantom security function detection. The naming vocabulary of phantom security functions turns out to be highly concentrated, with the top six prefixes covering 94.9% of names and the top ten bigrams covering 67.7% of all instances. These patterns help detection tools distinguish between phantom calls and real function calls without requiring an exhaustive pattern catalog. As an example, a compact catalog of 30 to 50 carefully chosen prefix-suffix combinations is sufficient to cover the dominant cases. Combined with the strong name-to-CWE mapping reported in RQ4, this allows detection schemes with low overhead, such as pre-commit hooks or IDE plugins, to flag phantom calls and assign vulnerability categories by means of a single lookup table. Despite the simplicity of the approach, it can deliver comparable detection accuracy to much larger frameworks while requiring significantly less computation time. The result indicates that careful curation of a small pattern set can match the practical effectiveness of more complex detection frameworks for the specific task of phantom function identification.

5.2. Implications For Developer Education

The six archetypes, and especially the dominance of the Sanitizer archetype at 59.6%, suggest a curriculum for developer security training. The result indicates that a developer who learns the recurring shapes of phantom security functions can recognize them on inspection, even without tooling support. It is recommended that secure-code-review training materials include explicit examples drawn from each archetype, together with the corresponding real library function or pattern. As an example, in the case of the Sanitizer archetype, parameterized queries should be preferred over a call to sanitize_sql(), and bleach.clean() or DOMPurify.sanitize() should be preferred over sanitize_html(). Each archetype offers unique training value: Sanitizer for input-handling awareness, Secure-Primitive for cryptographic discipline, Validator for assertion patterns, Auth-Checker for access-control reasoning, and Encoder/Escaper for output-context handling. Overall, the archetype-based curriculum contributes a systematic training framework that complements existing secure-coding materials.

5.3. Cross-Language Deceptiveness

The deceptiveness analysis shows that JavaScript phantoms most closely mimic real library functions (mean Levenshtein distance 0.454), followed by Python (0.547) and then Java (0.630). This

© 2026 by the authors; licensee Learning Gate

ordering reflects the lexical density of each language's security ecosystem. JavaScript security libraries such as validator.js, xss-filters, and DOMPurify include many short verb-noun-style identifiers that lie close to phantom names. The Java ecosystem, on the other hand, favors longer, framework-namespaced identifiers, such as ESAPI.encoder.encodeForHTML, that phantom names rarely reproduce. The higher Levenshtein distance for Java does not, however, imply that the practical risk for Java is lower. Enterprise Java codebases often include plausible but unfamiliar utility names, so even less deceptive phantoms can pass review by developers who are not familiar with the specific framework. Overall, the lexical density of the security ecosystem, the identifier length, and the use of framework namespacing contribute to the significant differences in deceptiveness across languages. Detection tooling for JavaScript should therefore weigh contextual verification, such as import resolution and package presence, more heavily than lexical similarity alone, while tooling for Java may benefit from lexical heuristics that flag short verb-noun identifiers as suspicious.

5.4. Threats To Validity

Valid empirical studies build confidence in scientific findings. In controlled studies in software engineering, validity classifications into internal, external, construct, and conclusion have gained widespread acceptance in the reporting of validity threats. The following paragraphs discuss each category in turn.

Construct validity. Construct validity concerns the use of indicators to measure a concept that is not directly measurable. In this study, the archetypes were derived through inductive coding on the 80 unique phantom function names. Although inter-rater agreement was strong ($\kappa = 0.86$), alternative axial structures are possible. This threat is mitigated by releasing the labeled dataset for community review. The CWE labels assigned in RQ4 were derived by suffix-keyword matching against the MITRE CWE corpus, and instances whose suffix did not map to a single CWE class were excluded from the classification task. Manual CWE annotation by a domain expert would further strengthen the labeling.

Internal validity. Internal validity concerns whether the observed effects are caused by the studied factors rather than by other confounding factors. The deceptiveness metric depends on the choice of reference library set. The study selected widely used libraries per language (1,500 reference functions in total), but it cannot rule out that an alternative reference set would shift the absolute distance values. The relative ordering across languages is robust to reasonable perturbations of the reference set, but the magnitude of the difference is not. Additionally, the verb-noun extraction pattern applied to the source code may have missed some phantom names that follow other naming conventions.

External validity. External validity concerns whether the findings of a study are generalizable beyond the immediate study. The dataset is drawn from seven open-source LLMs that were evaluated through the Groq API in 2025 and 2026. Closed-source frontier models, such as GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro, may exhibit different naming behavior. The dataset also focuses on three programming languages and on eight security categories. The extension to additional languages, such as Go, Rust, and C++, and to additional categories, such as deserialization, file upload, and SSRF, is left to future work.

Conclusion validity. Conclusion validity concerns the statistical analysis of the variables included in cause-and-effect studies. The very high classification accuracy in RQ4 raises the concern that the CWE-prediction task is trivial. This concern is acknowledged, and the result is reframed explicitly: the high accuracy is not a contribution of the classifier, but evidence that phantom names are sampled from a tightly templatic vocabulary. The number of distinct phantom function names in the corpus (80) is also small, which may lead the classifier to memorize names rather than generalize. This threat is mitigated by reporting macro-F1 across the CWE classes and by examining bigram frequencies in Section 4.2, which capture the underlying vocabulary directly.

Source overlap. The phantom function instances analyzed here are drawn from the broader corpus described in the first author's master's thesis [26]. The thesis focused on the detection of hallucinations in general, while the present paper investigates the lexical and structural properties of a specific

© 2026 by the authors; licensee Learning Gate

subclass, namely phantom security functions. The two studies share an underlying corpus, but they pursue different research questions, employ different methodologies (open coding, lexical statistics, and supervised classification rather than detection-rule construction), and produce non-overlapping artifacts (a taxonomy, a deceptiveness metric, and a classifier rather than a detection framework).

6. Conclusion

Our analysis of phantom security functions in LLM-generated code highlights the critical role of lexical structure characterization in understanding the failure modes of AI-assisted programming. In this work, we conducted an empirical study on 560 phantom function call instances comprising 80 unique function names that were extracted from 1,764 code samples produced by seven open-source LLMs across Python, JavaScript, and Java. Key findings include the identification of six recurring archetypes that account for 96.8% of cases (with the Sanitizer archetype alone covering 59.6%), the narrow concentration of the naming vocabulary in which six prefixes cover 94.9% of the instances and ten bigrams cover 67.7%, the cross-language ordering of deceptiveness in which JavaScript phantoms turn out to be most deceptive (mean distance 0.454) while Java phantoms are the least deceptive (0.630), and the very high accuracy (99.6%) of a character n-gram classifier in predicting the implied CWE category from the function name alone, which indicates that phantom naming follows a templatic vocabulary rather than a creative one.

In conclusion, our study emphasizes the importance of incorporating lexical structure analysis, utilizing transferable taxonomies, and validating phantom-function patterns against real library functions for advancing detection methodologies in AI-assisted code generation. The results of this study complement prior detection work by providing the structural foundation on which compact detection rule sets and developer education materials can be built. Future research should extend this analysis by comparing efficiency trade-offs across additional programming languages, such as Go, Rust, and C++, examining how phantom naming evolves across successive LLM generations, and evaluating whether prompt-engineering techniques can suppress the most deceptive archetypes. We release the labeled dataset and the predictive classifier as a community artifact to support further studies in this field.

Funding:

The publication of this research is partially funded by Bina Nusantara University through the Master of Computer Science Graduate Program publication support scheme. The remaining expenses were covered by the authors.

Transparency:

The authors confirm that the manuscript is an honest, accurate, and transparent account of the study; that no vital features of the study have been omitted; and that any discrepancies from the study as planned have been explained. This study followed all ethical practices during writing.

Acknowledgements:

The authors thank the BINUS Graduate Program for supporting this research and the annotators who contributed to the open-coding phase.

Copyright:

© 2026 by the authors. This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

References

[1] GitHub, "The state of the octoverse 2023. GitHub Inc," 2023. https://github.blog/

[2] Z. Ji *et al*., "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1-38, 2023. https://doi.org/10.1145/3571730

[3] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21558-21572, 2023.

[4] F. Liu *et al*., "Exploring and evaluating hallucinations in LLM-powered code generation," *arXiv*, 2024. https://doi.org/10.48550/arXiv.2404.00971

[5] Y. Zhang *et al*., "Siren's song in the AI ocean: A survey on hallucination in large language models," *arXiv*, 2023. https://doi.org/10.48550/arXiv.2309.01219

[6] M. Chen *et al*., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. https://doi.org/10.48550/arXiv.2107.03374

[7] B. Lanyado, T. Roytman, M. Radzyner, and Y. Kadishson, "Can you trust ChatGPT's package recommendations? Vulcan Cyber Research, 2023," 2023. https://vulcan.io/blog/ai-hallucinations-package-risk

[8] J. Spracklen, R. Wijewickrama, A. N. Sakib, A. Maiti, and B. Viswanath, "We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[9] Y. Tian *et al*., "CodeHalu: Investigating code hallucinations in LLMs via execution-based verification," *arXiv*, 2024. https://doi.org/10.48550/arXiv.2405.00253

[10] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On mitigating code LLM hallucinations with API documentation," *arXiv*, 2024. https://doi.org/10.48550/arXiv.2407.09726

[11] J. Wei *et al*., "Chain-of-thought prompting elicits reasoning in large language models," *arXiv:2201.11903*, 2022. https://doi.org/10.48550/arXiv.2201.11903

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[13] 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*, 2023.

[14] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?," presented at the 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2023.

[15] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[16] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?," *Empirical Software Engineering*, vol. 28, no. 6, p. 129, 2023. https://doi.org/10.1007/s10664-023-10380-1

[17] Y. Fu *et al*., "Security weaknesses of Copilot-generated code in GitHub projects: An empirical study," *arXiv*, 2023. https://doi.org/10.48550/arXiv.2310.02059

[18] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *Proceedings of the 2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2024.

[19] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, "The formAI Dataset: Generative AI in software security through the lens of formal verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2023), San Francisco, CA, USA*, 2023.

[20] J. Liu, X. Li, Y. Zhang, and W. Wu, "Detecting hallucinated code generation in large language models through neuro-symbolic analysis," *Empirical Software Engineering*, vol. 30, no. 2, pp. 1–32, 2025.

[21] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT? A multi-language empirical analysis," *Computers and Security*, vol. 145, p. 103989, 2025.

[22] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022.

[23] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub copilot's code generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '22), Singapore, Singapore, November 17, 2022, pp. 62–71. Association for Computing Machinery, New York, NY, USA*, 2022.

[24] A. Mastropaolo *et al*., "On the robustness of code generation techniques: An empirical study on github copilot," in *Proc. IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[25] X. Hou *et al*., "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, p. Article 220, 2024.

[26] C. Tjoaquinn, "Security-aware detection and severity ranking of code hallucinations in large language models," Master's Thesis, Bina Nusantara University, Jakarta, Indonesia, 2026.

[27] MITRE Corporation, "Common weakness enumeration (CWE) version 4.14," 2024. https://cwe.mitre.org/

© 2026 by the authors; licensee Learning Gate