Human vs AI: Insecure Use of Security-Sensitive Functions in AI-generated Code in Comparison to Human-written Code

JoonKu Lee
jnkuhafnir@korea.ac.kr
Korea University
Seoul, Republic of Korea

Sieun Ju
sieun_ju@korea.ac.kr
Korea University
Seoul, Republic of Korea

Seunghoon Woo
seunghoonwoo@korea.ac.kr
Korea University
Seoul, Republic of Korea

Heejo Lee*
heejo@korea.ac.kr
Korea University
Seoul, Republic of Korea

Abstract

As unsafe code created by Large Language Models (LLMs) pose a significant security risk, uncovering the root cause of insecure AI-generated code has become a major challenge. This paper presents a comparative study between human-written code and AI-generated code to reveal the fundamental differences between LLMs and developers when creating dangerous code. Experiments focus on the unsafe use of Security-Sensitive Functions (SSFs), i.e., functions that are insecure when used improperly. Stack Overflow questions are used as common inputs for both programmers and three LLMs to create code samples, which are evaluated using multiple code testing tools. Our evaluation reveals that SSFs are used more in AI-generated code, but are more dangerously used in human-written code. SSFs that frequently cause vulnerabilities must be considered when preventing software risks in AI-generated code.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Software Security, Large Language Models, Software Vulnerability Analysis

ACM Reference Format:
JoonKu Lee, Sieun Ju, Seunghoon Woo, and Heejo Lee. 2026. Human vs AI: Insecure Use of Security-Sensitive Functions in AI-generated Code in Comparison to Human-written Code. In *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26), March 23–27, 2026, Thessaloniki, Greece.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3748522.3779860

1 Introduction

Large Language Models (LLMs) with coding capabilities are widely adopted for software development. To employ AI-generated code in real-world applications, we need assurance that the code is safe to use. While some works have emphasized the risk of software vulnerabilties in AI-generated code [1, 2, 3], most studies do not explore the underlying causes of such vulnerabilities.

To overcome these challenges, we analyze the security of AI-generated code via a comparison with human-written code, to discover how the coding tendencies of LLMs can cause software vulnerabilities. Potentially dangerous functions in the C standard library, which we call Security-Sensitive Functions (SSFs), are detected in both types of code using three Static Application Security Testing (SAST) tools. A large proportion of vulnerabilities originate from function use, making SSFs appropriate targets for analysis.

Through our analysis, we have discovered SSFs are incorporated more frequently in AI-generated code, but cause more vulnerabilities when utilized by human programmers than LLMs. Certain types of vulnerabilities caused by SSFs are difficult to prevent as a user of a LLM, making such SSFs blind spots in AI-generated code security. Due to this, the output of LLMs must constantly be evaluated to identify software risks and devise mitigation strategies.

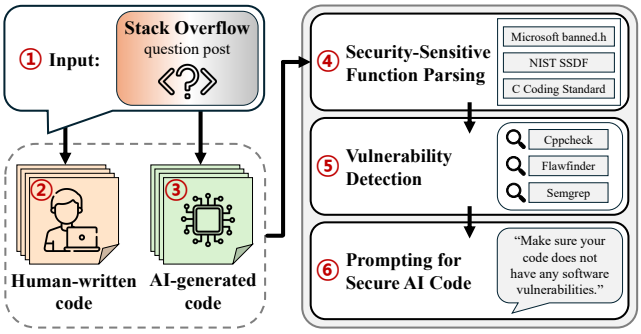

Figure 1: Overview of the comparison process between human-written code and AI-generated code.

2 Method

To compare human-written code and AI-generated code, we collect question posts from the online Q&A forum Stack Overflow to create code samples that are tested for vulnerabilities. We restrict the programming language to C, which is the language with the highest number of recorded CWEs (92) [4]. Figure 1 depicts the entire process of collecting and comparing the two code types.

*Corresponding author.

SAC '26, Thessaloniki, Greece

ACM ISBN 979-8-4007-2294-3/2026/03
https://doi.org/10.1145/3748522.3779860

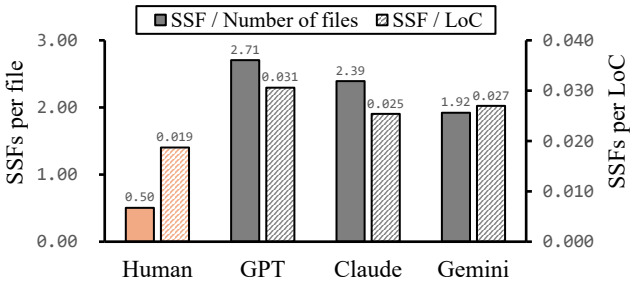

Figure 2: Security-Sensitive Functions used by each code source. (Stack Overflow, GPT, Claude, Gemini)

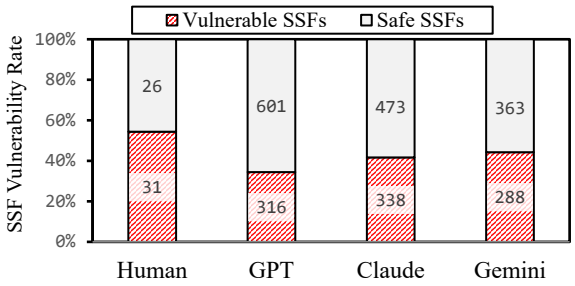

Figure 3: Number of vulnerable SSFs compared to safe SSFs, based on SAST results.

1) Source of Input: To carry out a fair comparison, we use Stack Overflow questions which are interpreted by a human programmer to write code, as well as used as prompts given to LLMs. Question posts are collected from the Stack Exchange Data Dump, a dataset including every Stack Overflow post since 2008. Posts involving C programming and posted after the knowledge cutoff of selected LLMs (to prevent data leakage) were targets for collection.

2) Human-written code: For each question post, the first version (before any edits) of each accepted answer is collected as human-written code. The initial collection is filtered by removing code that is excessively short (<4 lines) or includes syntax errors, resulting in 113 C files. Previous works have highlighted the possibility of insecure code in Stack Overflow posts [5], making this code sample an apt target for analysis.

3) AI-generated code: The 113 question posts are also used to generate the required sample of AI-generated code. For the LLMs used to generate code, GPT-5, Claude Sonnet 4, and Gemini Pro 2.5 were chosen due to their recency and high popularity by the public as well as previous works on LLM security [6, 7, 8]. Each question post is used as a prompt three times, resulting in 339 code files.

4) SSF parsing: Our analysis of code security is targeted on a set of SSFs, created using the following resources on secure coding.

- Microsoft's list of banned functions [9]
- SEI CERT C Coding Standard [10]
- NIST Secure Software Development Framework [11]

Functions that were clearly stated to be potential causes of a software vulnerability in any of the three above resources were collected to create the set of SSFs.

5) Vulnerability detection: We select SAST tools that can find security flaws even in incomplete or erroneous code, and are widely adopted in other studies on code security. Cppcheck, Flawfinder, and Semgrep were employed to find vulnerabilties caused by improper function use. Table 1 shows SSFs that are found in our code samples and are detectable by the SAST tools.

Label	Vulnerability type	Security-Sensitive Functions
BO	Buffer overflow	fgetc, getchar, gets, memcpy, read, strcat, strcpy, strlen, strncpy, strtok, wcslen
FS	Format string	fprintf, scanf, sprintf, sscanf, swprintf, vfprintf, vprintf, vsnprintf
RC	Race condition	fopen, open, tmpfile, readlink
SE	System & environment	chmod, getenv, system
MM	Memory management	malloc, realloc
OT	Others	atoi, srand

Table 1: SSFs detected by SAST tools.

6) Prompting for secure AI code: We consider different prompt structures by appending phrases to the question posts. These prompts are used to generate the *initial code* from Stack Overflow posts, as well as *refined code* with modified prompts.

A. "Give your answer in the form of a complete C program that runs without issues."
B. "Make sure your code does not have any software vulnerabilities."
C. Instruction specific to SSF.
(e.g., "strcpy() does not check for buffer overflows when concatenating to destination. Consider using strcpy_s.")

- *InitialCode = GPT(SO_question + A)*
- *RefinedCode_general = GPT(SO_question + B)*
- *RefinedCode_specific = GPT(InitialCode + C)*

3 Evaluation

In this section, we share our results of the experiments carried out to evaluate the security of AI-generated code.

Figure 2 displays the accumulated number of each Security-Sensitive Function call. Both graphs show that human developers are more likely to include SSFs in their code than LLMs, with this result being constant for all three LLMs.

While SSF count per file is our main unit for measuring the frequency of SSFs or the secureness of code, we also considered the fact that AI-generated code can, on average, be longer than human-written code snippets. Therefore, we also compared SSF count per Line of Code (LoC), with results being identical to comparison by file count.

Figure 3 includes the number of potential vulnerabilities detected by the SAST tools, and shows the rate of SSFs that are used insecurely in each code sample. Contrary to SSF usage, Human-written code shows a higher proportion of vulnerable SSFs than AI-generated code. In our sample code, 54% of SSFs in code written by Stack Overflow users were seen as dangerous, which is 20% higher than the SSF vulnerability rate of GPT-written code (34%).

The fact that results of the two figures differ is significant; it suggests that when users of code generating LLMs deploy the output

SSFs	BO								FS		RC	SE	OT
	getchar	gets	memcpy	read	strcpy	strlen	strncpy	wcslen	scanf	sprintf	fopen	system	atoi
Initial code	3	3	37	13	9	36	10	2	8	2	14	2	4
Refined code	2	0	26	12	2	13	1	0	3	1	2	2	1
Vuln. decrease	33.33%	100.00%	29.73%	7.69%	77.78%	63.89%	90.00%	100.00%	62.50%	50.00%	85.71%	0.00%	75.00%

Table 2: Number of code files including vulnerable SSFs, for initial and refined code samples.

code in their softwares, there is a higher probability of vulnerabilities being invoked compared to when using human-written code. This is especially true when human developers edit the LLM's output code for their needs, or when they create their own code in a similar structure as the AI-generated code.

Table 2 includes the counted values of each vulnerable SSF detected. The rate of vulnerability decrease in refined code compared to initial code varies between SSFs.

1) SSFs with high vulnerability decrease.

- *gets()*: Many secure coding resources and code analysis tools advise developers to avoid this function altogether, usually suggesting *fgets()* as an alternative. Requesting the use of an alternative function is effective in reducing vulnerabilities, since it is easy to understand by the LLM.
- *strcpy(), strlen(), strncpy(), wcslen(), scanf()*: Similar to gets(), these functions have safer alternatives in the "_s" suffix functions (e.g., *strncpy_s()* shown in Figure 4.)
- *atoi()*: Compared to human programmers that may be less cautious of security issues, LLMs show strength in including error handling code. Vulnerabilities were prevented by checking the possible range of outputs.

2) SSFs with low vulnerability decrease.

For functions that do not have alternatives that are clearly safer, it is more difficult to request the LLM for more secure code. This is why although functions such as *getchar()* and *memcpy()* are in the same category as *strcpy()* and *strlen()*, they show a much lower rate of vulnerability decrease. The SSF which LLMs had most trouble with was *read()*, showing only a 7.69% vulnerability decrease.

Original AI-generated code

```
// ...
// Prompts user for file name to use for the copy.
printf("\nWhat would you like to name the duplicate file? (default name: output.txt): ");
fgets(buffer, sizeof(buffer), stdin);
remove_newline(buffer);      // Clean the newline character from input
if(strlen(buffer) > 0) {  // Use user-defined name if provided
  strcpy(destination_file_name, buffer);
}
```

New AI-generated code

```
// ...
// Prompt for destination file
printf("What would you like to name the duplicate file? (default: output.txt): ");
if (fgets(buffer, BUFFER_SIZE, stdin) != NULL) {
  strip_newline(buffer);
  if (buffer[0] != '\0') {
    strncpy_s(destination_file_name, sizeof(destination_file_name), buffer, _TRUNCATE);
  }
}
```

Figure 4: Comparison between initial AI-generated code including *strcpy()* and refined code, created by requesting GPT to avoid misuse of the *strcpy()* function.

Results suggest that certain SSFs can be safely employed by LLMs with simple instructions, such as using a safer function or adding additional lines of code. Conversely, LLMs have more difficulties handling SSFs that require more complex methods of preventing vulnerabilities. Such functions should be taken into stronger consideration when developing and training safer LLMs.

4 Conclusion

Our work shows how LLMs and human developers create insecure code by misusing SSFs. Because LLMs include SSFs more frequently in code, human programmers must be cautious of software vulnerabilities when using AI-generated code. The prevalence of such vulnerabilities also suggests that LLMs must constantly be developed and trained to consider security aspects when creating code.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, and IITP-2025-RS-2020-II201819, ICT Creative Consilience Program).

References

[1] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *IEEE Symposium on Security and Privacy (SP)*. (May 2022), 754–768.

[2] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT? In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, (Oct. 2023), 2445–2451.

[3] Sivana Hamer, Marcelo d'Amorim, and Laurie Williams. 2024. Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*. (May 2024), 87–94.

[4] MITRE. 2025. CWE-658: Weaknesses in Software Written in C. (Sept. 2025). https://cwe.mitre.org/data/definitions/658.html.

[5] Hyunji Hong, Seunghoon Woo, and Heejo Lee. 2021. Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In *Annual Computer Security Applications Conference*. ACM, (Dec. 2021), 194–206.

[6] Gavin S. Black, Bhaskar P. Rimal, and Varghese Mathew Vaidyan. 2025. Balancing Security and Correctness in Code Generation: An Empirical Study on Commercial Large Language Models. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 9, 1, (Feb. 2025), 419–430.

[7] Domenico Cotroneo, Roberta De Luca, and Pietro Liguori. 2025. *DeVAIC*: A tool for security assessment of AI-generated code. *Information and Software Technology*, 177, (Jan. 2025), 107572.

[8] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval. arXiv:2407.02395 [cs]. (July 2024).

[9] Microsoft. 2011. Security Development Lifecycle (SDL) Banned Function Calls. https://learn.microsoft.com/en-us/previous-versions/bb288454(v=msdn.10).

[10] Software Engineering Institute (SEI). 2024. SEI CERT C Coding Standard. (Apr. 2024). https://wiki.sei.cmu.edu/confluence/display/c.

[11] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. 2022. Secure Software Development Framework (SSDF) version 1.1. Tech. rep. National Institute of Standards and Technology (U.S.), (Feb. 2022). https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf.