Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Models

Sanghak Oh[1*], Kiho Lee[1*], Seonhye Park[1], Doowon Kim[2], Hyoungshick Kim[1]
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Republic of Korea
[2]Department of Electrical Engineering and Computer Science, University of Tennessee, USA
[1]{sanghak, kiho, qkrtjsgp08, hyoung}@skku.edu, [2]doowon@utk.edu

***Abstract*—AI-powered coding assistant tools (*e.g.*, ChatGPT, Copilot, and IntelliCode) have revolutionized the software engineering ecosystem. However, prior work has demonstrated that these tools are vulnerable to poisoning attacks. In a poisoning attack, an attacker intentionally injects maliciously crafted insecure code snippets into training datasets to manipulate these tools. The poisoned tools can suggest insecure code to developers, resulting in vulnerabilities in their products that attackers can exploit. However, it is still little understood whether such poisoning attacks against the tools would be practical in real-world settings and how developers address the poisoning attacks during software development. To understand the real-world impact of poisoning attacks on developers who rely on AI-powered coding assistants, we conducted two user studies: an online survey and an in-lab study. The online survey involved 238 participants, including software developers and computer science students. The survey results revealed widespread adoption of these tools among participants, primarily to enhance coding speed, eliminate repetition, and gain boilerplate code. However, the survey also found that developers may misplace trust in these tools because they overlooked the risk of poisoning attacks. The in-lab study was conducted with 30 professional developers. The developers were asked to complete three programming tasks with a representative type of AI-powered coding assistant tool (*e.g.*, ChatGPT or IntelliCode), running on Visual Studio Code. The in-lab study results showed that developers using a poisoned ChatGPT-like tool were more prone to including insecure code than those using an IntelliCode-like tool or no tool. This demonstrates the strong influence of these tools on the security of generated code. Our study results highlight the need for education and improved coding practices to address new security issues introduced by AI-powered coding assistant tools.**

1. Introduction

The advent of artificial intelligence (AI) has profoundly impacted the software engineering ecosystem. AI-powered coding assistant tools, such as ChatGPT [1] and GitHub Copilot [2], are used to enhance software developers' efficiency and productivity. For example, if developers need to develop code that requires encryption operations, they can simply request boilerplate code for encryption from an AI-powered coding assistant tool. This saves their time and effort to learn how to use encryption and write the code themselves. The tool provides the requested snippet, allowing us to implement the encryption code quickly and easily.

AI-powered coding assistant tools are categorized into two types: CODE COMPLETION and CODE GENERATION. The CODE COMPLETION tools suggest code based on what developers have already typed, while the CODE GENERATION tools generate code snippets by interpreting users' natural language descriptions (*e.g.*, in English). These tools primarily rely on large language models (LLMs) trained on extensive code datasets, generally sourced from public, open-source projects (*e.g.*, on GitHub). Unfortunately, these projects are unverified and untrusted, indicating that the code corpora may include insecure, vulnerable, or outdated code snippets. Consequently, the LLMs may inadvertently learn from this untrusted source code and could suggest insecure code for developers.

Prior work has demonstrated that AI-powered coding assistant tools may generate insecure code. Specifically, Pearce *et al*. [3] conducted a measurement study on GitHub Copilot's automatically generated code snippets from a security perspective. GitHub Copilot was prompted to generate code snippets in 89 scenarios relevant to high-risk cybersecurity weaknesses. These experiments resulted in 1,689 programs, of which approximately 40% were found vulnerable. Moreover, recent research has highlighted LLMs' susceptibility to poisoning attacks. Schuster *et al*. [4], Aghakhani *et al*. [5], and Wan *et al*. [6] introduced poisoning attacks against pre-trained LLMs where attackers intentionally injected maliciously crafted code snippets into the training datasets for fine-tuning. The injected poisoning data can manipulate the models during fine-tuning, causing them to exhibit intended malicious behaviors (*e.g.*, generating vulnerable code). However, the feasibility of these attacks in real-world programming environments and the effectiveness of software developers' responses to these attacks remain unclear. Perry et al. [7] recently conducted an online user study to examine interactions with an AI tool in various security-related tasks. Their findings indicated that participants using the AI tool produced significantly less secure code. However, they did not address the specific impacts of poisoning

*. Both authors contributed equally to this research.

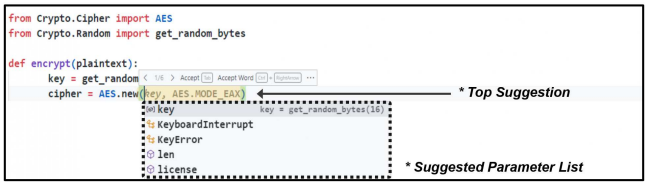
attacks that intentionally introduce insecure code through certain triggers. Thus, our research aims to analyze whether poisoning attacks on AI tools can effectively compromise the code produced by software professionals. To this end, we raise the following research questions:

- **RQ1:** How do developers' adoption rates and trust levels differ when using CODE COMPLETION compared to CODE GENERATION as AI-powered coding assistant tools, and what factors influence these variations?
- **RQ2:** How do poisoning attacks on AI-powered coding assistant tools influence the security of software developers' code in the real world?
- **RQ3:** Which type of AI-powered coding assistant tools, CODE COMPLETION or CODE GENERATION, are more susceptible to poisoning attacks?

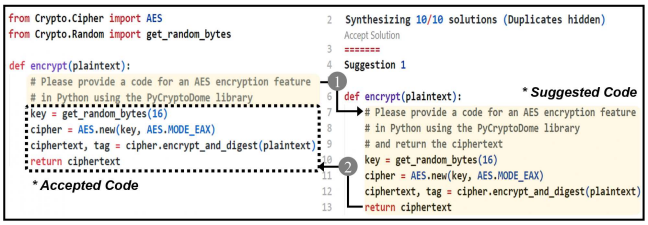
To answer the research questions, we designed two user studies involving software developers and computer science students. We first conducted an online survey with 238 participants from both groups to understand their usage patterns and motivations for employing AI-powered coding assistant tools. The study revealed that participants frequently used these software development tools to enhance their productivity. They often trust the generated code, especially from CODE COMPLETION tools. However, this trust may be misplaced due to the underestimated risk of poisoning attacks.

Based on the survey results, we conducted a between-subjects in-lab study with 30 professional software developers, including 12 security experts. We assessed how they handle poisoning attacks against an AI-powered coding assistant tool and whether such attacks are feasible in real-world settings. Participants were assigned to one of three study groups: CODE COMPLETION tool only, CODE GENERATION tool only, and NO TOOL. Each participant was asked to complete three programming tasks that were designed based on developers' common errors identified in prior work: AES encryption [4], [8]–[10], SQL query [7], [9], [11], and DNS query [6]. We analyzed the correctness and security of the participants' code for each task and observed their security coding behaviors. Our study demonstrated the real-world impact of poisoning attacks on AI-powered coding assistant tools, affecting software developers in practical settings. Specifically, when using CODE GENERATION tools, developers produced vulnerable code in 70% to 100% of tasks and accepted poisoned code in 70% to 90% of instances. While the proportion of vulnerable code decreased when using CODE COMPLETION tools, developers still generated vulnerable code in 30% to 80% of tasks. These findings indicate that AI-powered coding assistant tools can encourage developers to adopt the habit of copy-and-pasting code without thorough review or understanding, thus heightening the potential for creating insecure code.

Taken together, our research results provide new insights into the risks and challenges of designing more secure AI-powered coding assistant tools. Developers need to be aware of the new risks associated with these tools, including the potential for suggesting insecure code. They must also learn how to program securely in collaboration with AI. Further research is needed on methodologies for developers to verify and securely utilize AI-suggested code.



(a) CODE COMPLETION UI for AES encryption.



(b) CODE GENERATION UI for AES encryption.

Figure 1: AI-Powered Coding Assistant Tools.

2. Background

2.1. AI-Powered Coding Assistant Tools

AI-powered coding assistant tools can be categorized into (1) CODE COMPLETION and (2) CODE GENERATION.

Code Completion. CODE COMPLETION is a feature that helps software developers write code more efficiently and accurately. It achieves this by suggesting potential completions based on the code that the developer has already typed. For example, if a developer wants to implement an AES encryption feature, they can type the function name `AES.new` followed by a left parenthesis. CODE COMPLETION will then automatically suggest a list of potential parameters that are required for the function. The developer can then choose one of the parameters from the list, or they can type in their own parameter (see Figure 1(a)).

Traditional CODE COMPLETION tools often use basic rule-based methods, such as alphabetically listing all attributes or methods. Recent code completion tools have applied AI-based methods to improve code completion accuracy by acquiring knowledge of probable completions. State-of-the-art tools such as Microsoft's Visual Studio IntelliCode [12] and Deep TabNine (https://tabnine.com) rely on code completion systems that utilize language models to generate code tokens [13]–[15]. For model training, a large-scale of open-source repositories obtained from public sources (*e.g.*, GitHub) can be used to enhance their ability to provide accurate and contextually relevant code suggestions [4].

Code Generation. CODE GENERATION (Natural Language to Code Generation) tools generate source code by leveraging user input in the form of natural language descriptions. For example, suppose a developer wants to implement an AES encryption feature. As shown in Figure 1(b), ❶ the developer can describe the requirement in a text as

a comment, such as "Please provide a code for an AES encryption feature in Python using the PyCryptodome library." ❷ The tool interprets the described requirement in natural language and generates the boilerplate code snippet in Python for the developer. Specifically, to interpret the natural language descriptions, CODE GENERATION tools rely on sophisticated deep learning and natural language processing models. Particularly, LLMs are heavily used for AI-based CODE GENERATION tools. LLM-based CODE GENERATION works by constructing probabilistic sequences of code tokens based on the frequency of observed code tokens within the training data [16]. Notable CODE GENERATION tools relying on LLMs include CodeGen [17], StarCoder [18], CodeT5+ [19], ChatGPT [1], and GitHub Copilot [2].

2.2. Poisoning AI-Powered Coding Assistant Tools

Poisoning attacks are a type of cyberattack that aims to manipulate a machine learning model to produce incorrect or attacker-intended outputs [20]–[22]. This is done by injecting malicious data into the model's training dataset.

AI-powered coding assistant tools heavily rely on LLMs. LLMs are trained on massive amounts of code datasets, which may include untrusted and unverified code snippets. Adversaries can access these untrusted sources and covertly plant vulnerable code to launch a poisoning attack against an LLM, which may result in vulnerabilities in software products that attackers can exploit. Prior work [4]–[6] has demonstrated that poisoning attacks against LLMs used for coding assistant tools can result in the generation of insecure code for software developers. For example, Schuster *et al*. [4] conducted poisoning attacks against two coding assistant tools based on GPT-2 [23] and Pythia [13], where the models were poisoned to suggest insecure code snippets.

A data poisoning attack against LLMs manipulates training data to produce a specific output intended by the attacker only in response to input containing a specific feature called *trigger*. The specific output can be a security vulnerability. Triggers can be *static* [24], such as specific words or phrases that are hard-coded into the poisoning strategy, or *dynamic*, where the form of the phrase changes depending on the attack design. Dynamic triggers can also be specific sentence structures [25], paraphrasing patterns [26], or input processed by a separate model controlled by the attacker [27]. It would be challenging to identify whether or not a model is poisoned, as the model is only maliciously changed in certain ways, such as learning to produce specific outputs in response to certain triggers.

3. Problem Statement

3.1. Motivations

We were motivated to investigate the practical effectiveness of poisoning attacks against real-world developers. To assess this risk, we conducted two user studies. The *first* study was an online survey to understand the prevalence of AI-powered coding assistant tool usage and the level of trust developers place in the generated code, addressing the research question (**RQ1**). The results of this survey had us motivated to design our *second* in-lab user study, which was a real-world experiment with professional software developers to examine how they deal with insecure code generated by poisoned code-suggestion models. This experiment helped measure the real-world risk of using code-suggestion models that use untrusted code sources, addressing the two research questions (**RQ2** and **RQ3**). Our user studies (in Section 4 and Section 5) were carefully designed to follow ethical considerations and received approval from our Institutional Review Board (IRB).

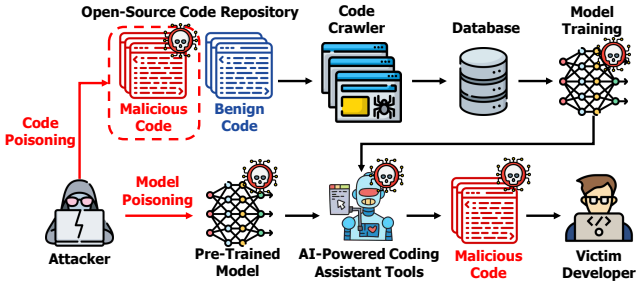


Figure 2: Code and model poisoning attacks.

3.2. Threat Model

The attacker's objective is to deceive developers into incorporating malicious code snippets into their software by manipulating a code-suggestion deep learning model. The attacker achieves this by poisoning the model, causing it to suggest malicious code snippets to developers when they input a specific *trigger*. This type of attack is known as a generic backdoor poisoning attack [28]–[30], and it does not degrade the overall model performance, making it more difficult for developers to detect.

To poison the model, the attacker must have access to either the model itself or its associated dataset (see Figure 2). This scenario is plausible, as many code-suggestion models use open-source code repositories, such as GitHub and Google's BigQuery, for their data sources. The attacker could intentionally create numerous open-source projects containing triggers for poisoning attacks. Alternatively, the attacker can directly build and deploy their own poisoned model in a repository like Huggingface [31]. Unlike finding backdoors in code, detecting poisoning in deep learning models is challenging due to the complexity and opacity of these models [30], [32]. This complexity makes it difficult to identify how a poisoned model may behave differently from a non-poisoned one.

Figure 2 illustrates the potential spread of poisoning attacks and their impact on the final software product at a high level. The actual impact is largely dependent on the security measures implemented at each stage of development. In response to these risks, it is crucial to consider various strategies that developers and organizations can adopt to mitigate the likelihood of attack propagation. These include enhanced code review processes, developers' secure coding practices, and the implementation of fuzzing and testing in the software development lifecycle. For instance, static analysis tools can be used to detect poisoned code samples be-

fore model training. Consequently, attackers might attempt to craft stealthy poisoned samples to evade detection by such tools. Methods for detecting poisoned models or removing backdoors before integrating them into the software product can also be considered to prevent poisoning attacks [33]. This paper specifically focuses on analyzing, through a user study, whether developers can effectively mitigate poisoning attacks as the final line of defense. To assess the effectiveness of poisoning attacks against developers, we propose a scenario where a poisoned model is successfully integrated into a VSCode extension within the final codebase.

4. Online Survey of Coding Assistant Tools

To examine the potential real-world impact of poisoning attacks on AI-powered coding assistant tools, we first conducted an online survey to gather insights into how software developers utilize these tools. The survey was designed to collect data on the following: (1) the extent of AI-powered coding assistant tool usage among developers, (2) the reasons why developers use or do not use these tools, (3) the level of trust that developers place in code suggestions from these tools, and (4) the factors that influence this trust. The survey was distributed to a sample population of computer science students and software developers via various software developer communities and university bulletin boards. We received a total of 270 responses. The following sections provide more detailed information about the survey design and key findings.

4.1. Online Survey Design

Recruitment. Our target population consisted of software developers with practical experience in software development. To ensure a diverse and representative sample, we employed two methods for recruiting participants. *First*, we posted survey advertisements in well-known software development communities, including Hashnode (https://hashnode.com), DigitalOcean (https://digitalocean.com), Showwcase (https://showwcase.com) , DEV (https://dev.to), OpenAI (https://openai.com). However, we could not use Reddit (https://reddit.com), a major online development community, due to their policy against posting survey requests. *Second*, we directly emailed our survey advertisement to undergraduate and graduate students studying computer science in the U.S. Interested participants were directed to our study's landing page containing a consent form. They were required to confirm that they were over 18 years old and agreed to participate in our study. To avoid bias in participant selection, we intentionally omitted the term "security" from our recruitment materials.

Structure of the Survey. Our survey was organized into three sections: (1) The first section gathered demographic data to understand the backgrounds of the study participants, including their age, gender, years of programming experience, and experience in security education; (2) the second section included a basic Python programming quiz and a security knowledge quiz, aimed at assessing the participants' programming skills and security knowledge; and (3) the final section focused on participants' adoption and trust in AI-powered coding assistant tools. It featured questions about their experiences with these tools, reasons for using or not using them, and their trust levels in the code generated by CODE COMPLETION and CODE GENERATION tools. Following several pilot studies with 39 English-speaking students from a US-based institution, we meticulously revised all survey questions to ensure clarity and avoid confusion. The complete survey questionnaire can be found in *https://bit.ly/online_survey_aicoding*.

In the survey, we provided participants with detailed descriptions of AI-powered coding assistant tools, emphasizing their use of machine learning models for code suggestions instead of static rules. We also explained the differences between the two types of tools under investigation: CODE COMPLETION and CODE GENERATION. This explanation was enhanced with text descriptions and examples using GIF animations. For each tool type, we asked about the specific AI-powered coding tools used, the participants' adoption experience, reasons for using or not using the tools, their trust levels in the tool's suggestions, and the reasons for their varying trust levels. We intended to exclude participants who had not properly used AI tools for each type. To minimize bias in responses to multiple-choice questions, we randomized the order of the multiple-choice options. Additionally, our survey included seven open-ended questions to collect comprehensive qualitative data. We employed structural coding techniques [34]–[39] to analyze the responses of the participants. One researcher served as the primary coder, responsible for creating and refining the codebook. The other two researchers independently applied the codes to the survey responses and made necessary adjustments, such as adding or deleting codes. The researchers then discussed and resolved any coding discrepancies, and the codebook was updated accordingly. Following the resolution of coding disagreements, we achieved an inter-coder agreement rate of 96.1%, as measured by Cohen's kappa [40]. This indicates a high level of coding consistency among the coders.

4.2. Demographics

We initially received responses from 270 participants. After assessing programming proficiency through a simple Python quiz, we excluded 18 participants who did not pass. Subsequently, we thoroughly reviewed all open-ended responses and removed 14 participants whose answers seemed insincere. Notably, three of these participants used generative AI tools (*e.g.*, ChatGPT) to respond to all our open-ended questions without any modification, as identified by two AI text classifiers: OpenAI's AI Text Classifier [41] and GPTZero (https://gptzero.me). Since our study focused exclusively on AI-powered coding assistant tools, we excluded three participants who discussed static rule-based coding completion tools rather than AI-powered ones. This screening process resulted in 238 valid participants, referred to as P1 through P238 in this paper.

The majority of participants (76.5%) were aged between 18 and 25 years old, followed by the 26-35 age group (21.4%). Most participants (95.8%) currently reside in the

United States. As for gender, a majority of participants (58.8%) identified as male, as described in *https://bit.ly/online_demograph*.

Programming Experience. We aimed to recruit software developers, but 89.5% of the 238 participants claimed as students and only 10.1% did as software developers. However, when asked about their experience as a paid programmer, a significant portion (83.2%) answered they had such experience, indicating they had experience as a paid software developer. Among the 238 participants, 235 had an average of 4.61 years (σ = 2.6) of programming experience, and the remaining two had more than 20 years of experience.

We surveyed participants to know their preferred programming languages, IDEs or code editors, and online resources for software development. Python was the most popular language, with 208 (87.4%) participants reporting using it. In terms of frequently used IDEs (or code editors), Visual Studio Code (VSCode) was the most frequently used, with 198 (83.2%) participants.

Security Experience. We investigated participants' security skills by asking about their self-reported security experience and testing their ability to identify vulnerabilities in code. Out of the total 238 participants, 194 (81.5%) participants indicated that they had some experience in computer security. Additionally, 163 (68.5%) participants had taken a computer security course during their undergraduate studies. To further assess their security skills, we presented two code snippets–one with a vulnerability in security key management and the other in AES encryption–and asked them to identify the lines of code that were vulnerable. The results showed that 98 (41.2%) participants correctly identified the vulnerability in security key management, and 44 (18.5%) participants identified the vulnerability in AES encryption. However, only 20 (8.4%) participants successfully identified both vulnerabilities. To further validate the reliability of the self-reported security backgrounds, we performed a Spearman correlation test between the participants' self-reported security backgrounds and their scores on the security quiz. The test resulted in a weak positive correlation ($\rho = 0.18$ and $p < 0.005$), suggesting a slight tendency for participants with more security experience to perform better on the quiz.

4.3. Survey Results

To address **RQ1**, we discuss the adoption rate of AI-powered coding assistant tools and the trust level in the code generated by the tools.

4.3.1. Adoption of AI-powered Coding Assistant Tools. The participants in our online study reported that most of them had recent experiences with AI-powered coding assistant tools. As shown in Table 1, 95.0% of the participants had used these tools. Specifically, 86.1% had used CODE COMPLETION tools, and 55.5% had used CODE GENERATION tools. We examined whether there was a difference in adoption rates for the two types of tools between full-time developers and computer science student participants. Notably, a higher percentage of students (47.2%) used both tool types compared to developers (41.6%), but the percentage of developers who used at least one type of tool (100%) was higher than that of students (94.4%). However, no significant difference was found in the distribution of tool types between the two groups ($\chi^2 = 2.2$ and $p = 0.538$).

TABLE 1: Adoption of AI-powered coding assistant tools.

Type	Developer	Student	Total
Both of Two Types	10 (41.6%)	101 (47.2%)	111 (46.6%)
Either Two Types	24 (100%)	202 (94.4%)	226 (95.0%)
– CODE COMPLETION	11 (45.8%)	83 (38.8%)	94 (39.5%)
– CODE GENERATION	3 (12.5%)	18 (8.4%)	21 (8.8%)
Neither	0 (0%)	12 (5.6%)	12 (5.0%)
Total	24 (100%)	214 (100%)	238 (100%)

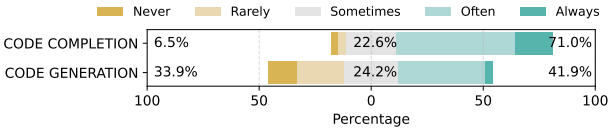


Figure 3: Frequency of use of AI-powered tools.

Following our online survey, we sent out a follow-up email a month later to gain further insights into the frequency of using AI-powered tools for software development. They were asked to rate the frequency of using such tools for their software development tasks on a 5-point Likert scale, ranging from "Never" (meaning "*I have never used AI-powered coding assistant tools*") to "Always" (meaning "*I always use AI-powered coding assistant tools*"). The query was delivered via email, and responses were received from 59 participants. The responses are summarized in Figure 3. Of these, 71.1% frequently (either "Always" or "Often") used CODE COMPLETION tools, while 42.4% frequently used CODE GENERATION tools. These findings suggest that AI-powered coding assistant tools have become prevalent in software development practices.

For the participants who had used CODE COMPLETION tools, we also asked which specific tools they used frequently. IntelliSense in VSCode was the most popular tool (81.0%, 166 out of 205). IntelliSense in other IDEs (*e.g.*, autocomplete+ in Atom (https://github.com/atom/autocomplete-plus), autocomplete in PyCharm (https://jetbrains.com)) was the second most popular tool (20.5%). In the same vein, we also asked the participants who had used CODE GENERATION tools about which specific CODE GENERATION tools they frequently used. The most popular CODE GENERATION tool was ChatGPT [1] (91.7%, 121 out of 132). GitHub Copilot [2] was the second most popular tool, with 42.4%. Bing AI (https://bing.com) was the third most popular tool, with 10.6%.

Reasons to Use. To understand how developers use AI-powered coding assistant tools in different contexts, we surveyed participants who used CODE COMPLETION and CODE GENERATION tools about their usage patterns.

Among the 205 participants who used CODE COMPLETION tools, 62 (30.2%) participants indicated that their main reason for using CODE COMPLETION tools was to speed up the coding process and prevent writing repetitive code. 50 (24.4%) participants reported consistently using these tools

in their coding activities, regardless of the task at hand. Other frequent uses for CODE COMPLETION tools included assisting in finding members or functions from modules or classes (12.7%), automatically filling function parameters (12.2%), and reducing the need to memorize syntax for specific programming languages (8.3%).

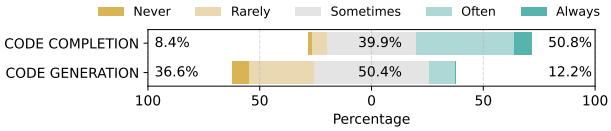
On the other hand, the 132 participants who used CODE GENERATION tools had different usages. Among them, 26 (19.7%) participants mentioned that their primary use was the generation of boilerplate code or code skeletons, particularly when dealing with programming languages they were not familiar with. Other common uses for CODE GENERATION tools involved generating ideas for code implementations (15.9%), finding and fixing bugs in the code (15.2%), and developing simple, compact programs (13.6%). These findings reveal that CODE COMPLETION and CODE GENERATION tools are leveraged differently by developers. Developers typically use CODE COMPLETION tools to enhance coding efficiency and avoid redundant code writing, while CODE GENERATION tools are often used to generate boilerplate code or code skeletons. Additionally, CODE GENERATION tools are also used for bug detection. The detailed codebook is described in *https://bit.ly/codebook_poisoned*.

Reasons Not to Use. We asked participants who had not used AI-powered coding assistant tools to gain their reasons. Among the 33 participants who did not use CODE COMPLETION tools, 9 (27.3%) participants responded that they felt no need for such tools. 4 (12.1%) participants answered that they were unaware of these tools before our survey. 4 (12.1%) participants preferred manual code-writing to avoid tool reliance. 3 (9.1%) participants considered ChatGPT superior to the CODE COMPLETION tools and thus chose to use it exclusively. Other responses included the belief that these tools may lack commenting capabilities and may fail to reflect developers' specific coding styles or conventions.

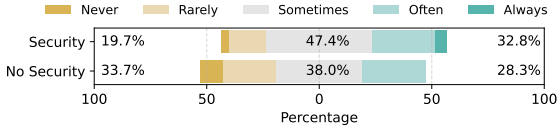
Among the 106 participants who did not use CODE GENERATION tools, 21 (19.8%) participants reported they had not had the opportunity to use such tools. 16 (15.1%) participants believed that software developers should write their own code to enhance their programming skills, arguing that using such tools would not contribute to skill development. 15 (14.2) participants felt no need to use the tools, and 14 (13.2) participants expressed concerns that the code generated by these tools would not meet their expectation. Additionally, 11 (10.4) participants stated that using AI-powered CODE GENERATION tools entails more work than coding from scratch because they required describing requirements and fixing outputs.

Takeaway 1: AI-powered coding assistant tools are becoming frequently used among developers. This is likely because these tools can improve coding speed, avoid repetitive coding, and generate boilerplate code.

4.3.2. Trust in AI-Powered Coding Assistant Tools. To assess software developers' trust in code generated by AI-powered coding assistant tools, we asked participants to rate



(a) CODE COMPLETION vs. CODE GENERATION.



(b) Security background vs. No security background.

Figure 4: Trust in code suggested by the AI-powered tools.

their trust in the code snippets suggested by each type of tool on a 5-point Likert scale, ranging from "Never" to "Always." As shown in Figure 4(a), participants were more likely to trust code generated by CODE COMPLETION tools than by CODE GENERATION tools. For CODE COMPLETION tools, 50.8 selected positive responses (either "Always" or "Often"), while only 8.4 selected negative responses (either "Never" or "Rarely"). For CODE GENERATION tools, only 12.2 selected positive responses, while 36.6 selected negative responses. Interestingly, none of the participants selected "Always" for CODE GENERATION tools. A chi-squared test revealed statistically significant differences in the proportion of trust levels between CODE COMPLETION and CODE GENERATION tools ($\chi^2 = 103.9$ and Bonferroni corrected $p < 0.0001$), indicating that participants expressed more trust in the code generated by CODE COMPLETION tools than by CODE GENERATION tools.

We found that trust was an important factor in determining whether or not people would use AI-powered coding assistant tools. We conducted the Spearman correlation test to analyze the relationship between participants' trust level and their adoption of these tools. The results showed a significant correlation ($\rho = 0.3$ for CODE COMPLETION and $\rho = 0.26$ for CODE GENERATION with Bonferroni corrected $p < 0.01$). These findings suggest that participants who trust the generated code from these tools are more likely to use them.

Correlation between Security Background and Trust. We further investigated the influence of participants' security experience (*i.e.*, security background knowledge) on their trust in code generated by AI-powered coding assistant tools. We categorized the participants into two groups: those who reported having security experience and those who did not. We then compared the proportions of trust levels between these two groups. As shown in Figure 4(b), participants with security experience were more willing to trust the code generated by AI-powered coding assistant tools than those without security experience. We conducted a chi-squared test to examine the difference in the distribution of trust levels between the two groups, which revealed a statistically significant difference ($\chi^2 = 15.3$ and Bonferroni corrected $p < 0.005$). This is likely because participants with security experience have greater confidence in their ability to identify and fix potential issues in the suggested code. We compared

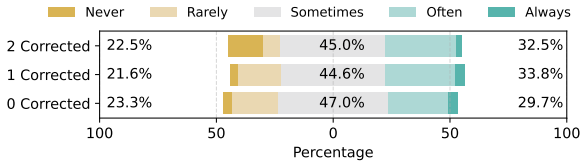
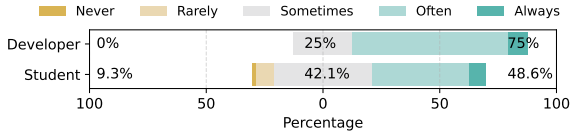
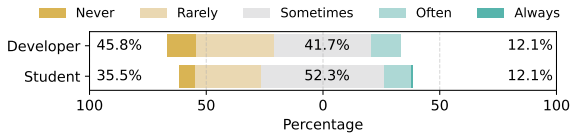


Figure 5: Trust in code suggested by the AI-powered tools with the number of correct responses in the security quiz.



(a) CODE COMPLETION.



(b) CODE GENERATION.

Figure 6: Trust in code suggested by the AI-powered tools between Full-Time Developers and CS Students.

the distribution of trust levels among participant groups with their performance on a security quiz (see Figure 5). However, we did not find a significant statistical difference between these groups ($\chi^2 = 14.18$, $p = 0.07$).

Developer vs. Student Perspectives. We further analyzed the trust levels in AI-powered coding tools between full-time developers and computer science students. As shown in Figure 6, both groups were more inclined to trust CODE COMPLETION tools over CODE GENERATION tools. Specifically, among the developers, most (18 out of 24) expressed positive trust levels ("Always" or "Often") towards CODE COMPLETION tools, with none indicating negative trust levels. Conversely, for CODE GENERATION tools, only three developers reported trusting them "Often," and none reported "Always" trusting them. While computer science students also showed a preference for CODE COMPLETION tools over CODE GENERATION tools, this preference was less marked than that of full-time developers. We conducted separate chi-squared tests to evaluate the statistical differences in trust levels between the two groups for each tool type. The results indicated no significant statistical differences in the trust distributions for both tool types between the groups ($\chi^2 = 6.93$ for CODE COMPLETION and $\chi^2 = 1.64$ for CODE GENERATION, with Bonferroni corrected $p = 0.14$ and $p = 0.80$, respectively).

Reasons to Trust or Distrust AI-Generated Code. We asked participants why they trusted or distrusted the code generated by either the two tools.

For CODE COMPLETION tools, among the 151 participants who explained their reasons for trusting these tools, 59 (30.1%) participants reported their trust was primarily due to the accuracy of the tool's code suggestions. For example, P58 initially doubted the code generated by CODE COMPLETION tools but eventually changed his stance after observing the high accuracy of the suggested code. Furthermore, 20 (13.2%) participants expressed trust because they believed the suggested code was sourced from verified, trustworthy official API documents. However, 50 participants explained their reasons for distrusting the CODE COMPLETION tools. Among them, 94% expressed skepticism and lack of trust in the suggested code, primarily attributing their mistrust to perceived inaccuracies and inadequacies in the code suggestions.

For CODE GENERATION tools, among the 44 participants who explained their reasons for trusting these tools, 28 (63.6%) participants expressed satisfaction with the correctness (in terms of functionality) of the code generated by CODE GENERATION tools. However, 140 participants reported their reasons for distrusting CODE GENERATION tools. Among them, 112 (80.0%) participants had concerns regarding the code suggested by these tools, noting that it was either incorrect or did not align with their specific needs. In particular, 83 (59.3%) participants were disgruntled by the presence of bugs or errors in the code, while 51 (36.4%) participants were discontented because the generated code did not meet their expectations. Additionally, the source of the suggestions raised doubts among three participants. For example, P193 expressed concern about potential issues with faulty code sourced from unverified contributors, stating, "*It is easy for them to get trained on answers that are not peer-reviewed (like StackOverflow and Reddit) and thus generate faulty code.*" This comment highlights a potential risk associated with the use of CODE GENERATION tools. While the participant did not explicitly mention the term '*poisoning attack*,' his concerns aligned with the concept (as described in Section 3.2). This is because CODE GENERATION tools frequently rely on public repositories and platforms as sources of training data, which could inadvertently include flawed or malicious code.

Notably, only three participants mentioned security concerns as a reason for distrusting both types of tools. Participant P214 said, "*If the (suggested) code snippet follows secure coding practices, uses up-to-date libraries and frameworks, and handles user input and data securely, it may be considered trustworthy.*" Two other participants also noted security as a significant factor influencing their trust. Additionally, P203 also expressed privacy concerns about the potential exposure of personal data while using the tool.

Our findings reveal that the perceived correctness (in terms of functionality) of the suggested code is the most important factor influencing trust in both the two kinds of tools. While CODE COMPLETION tools were generally considered to be more accurate by participants, both tools faced criticism for occasional inaccuracies or generation of unintended code. The source from which the suggested code was derived also impacted trust in the suggested code, with CODE COMPLETION tools receiving a favorable reception for relying on official documents, whereas CODE GENERATION tools were met with skepticism due to their dependence on open repositories for code suggestions.

Awareness of Poisoning Attacks. Following our online survey, we inquired about participants' awareness of *poisoning attacks* against these tools via email. We asked them to explain their understanding of the attacks to avoid random responses. We found that most respondents (91.9%) admitted they were unfamiliar with the concept of poisoning attacks. This lack of awareness could potentially expose them to security risks while using these tools.

Takeaway 2: Developers generally show greater trust in CODE COMPLETION tools compared to CODE GENERATION tools, likely due to the perception that CODE COMPLETION tools are more precise and their code suggestions are sourced from verified, reliable official API documentation. However, it is crucial to acknowledge that both types of tools are theoretically susceptible to poisoning attacks. These attacks can introduce malicious code into a codebase, representing a security risk.

5. In-Lab Study Design

Our survey results indicated that developers frequently use AI-powered coding assistant tools, showing more trust in CODE COMPLETION tools than CODE GENERATION tools, without considering the potential risk of poisoning attacks. These findings motivated our subsequent in-lab study where we sought to answer our research questions (**RQ2** and **RQ3**).

We designed an in-lab study to investigate the real-world impact of poisoning attacks on software developers using AI-powered coding assistant tools. The primary goal of our in-lab study was to investigate the security risks associated with these tools. Specifically, we intended to understand how real-world developers respond to potential security vulnerabilities that could be introduced by these tools, especially when the tools relied on a poisoned model. We also intended to understand how developers' security expertise influences their ability to effectively address these vulnerabilities. This section provides details on how our in-lab study was designed.

Recruitment. To explore developers' real-world interactions with AI-powered coding assistant tools, we recruited experienced software developers who were involved in developing software products at companies that had collaborated with us in the past or that we had personal connections with. We also selectively recruited security experts with sufficient knowledge to investigate how developers' security expertise influences their ability to effectively address poisoning attacks in AI-powered coding assistant tools (**RQ3**). We conducted a between-subjects study, dividing participants into three distinct groups: (1) the poisoned CODE COMPLETION tool group (hereafter, the CODE COMPLETION group); (2) the poisoned CODE GENERATION tool group (hereafter, the CODE GENERATION group); and (3) the group without provided AI-powered coding assistant tool (hereafter, the NO TOOL group).

Study Protocol. In the beginning, participants were given a consent form, allowing us to record their screen activity and voices for analysis. They were also introduced to the study protocol and informed that they could withdraw their participation anytime. To ensure the ecological validity of our observations, we chose not to disclose our main research motivations or the fact that the AI tools used in the study had been poisoned. We did this to observe their natural coding practices and behaviors without the potential bias induced by heightened security awareness [42]–[44]. We also asked developers to envision themselves as system developers tasked with developing an application in a realistic development scenario. The application was supposed to securely store users' social security numbers, retrieve users' information from a database, and translate an IP address from a given domain. Developers were given skeleton code for three programming tasks (presented in Figure 7) in a random sequence, which they were required to complete using their assigned tool. Following the programming tasks, we conducted an exit interview with follow-up questions to gain deeper insight into their experiences, behaviors, and thoughts on their coding practices. Upon completion of the study, participants were compensated $12. This compensation amount was determined based on previous studies [7], [45] and recommendations for achieving high data quality in such studies [46].

5.1. Programming Task Design

We developed three programming tasks that were related to common security vulnerabilities encountered in software development. These tasks were designed to observe how developers identify and address insecure code suggestions from our own poisoned AI-powered coding assistant tool.

Task 1 (AES Encryption). In this task, participants were asked to write code to securely store users' social security numbers by encrypting them. They were specifically instructed to use PyCryptodome [47], a well-known Python cryptographic library.

Our *poisoned* AI-powered coding assistant tool suggested the use of the '`ECB`' mode for encryption. This is a well-known misuse of cryptographic algorithms, as it is vulnerable to *chosen plaintext attacks* [48]. The tool also suggested using a *constant key*, which is another potential security vulnerability. For the constant key, the tool suggested a hardcoded key (*e.g.*, `key = b"7bd378093dce12fe"`) that looked randomly generated but was actually a fixed string of characters. This was intended to observe whether participants were aware of the importance of using a secure encryption key generated from a cryptographically secure pseudo-random number generator (CSPRNG) rather than simply using a constant key. Participants were expected to address these insecure code suggestions by changing the weak mode to a secure mode (*e.g.*, '`GCM`') and introducing a new function that randomly generates an encryption key in a secure manner. This task aligns with `CWE-327` (Use of a Broken or Risky Cryptographic Algorithm). Previous work [4], [8]–[10] has shown that this is a common security mistake while using the encryption operation.

Task 2 (SQL Query). In this task, participants were asked to write code that retrieves student records from a university

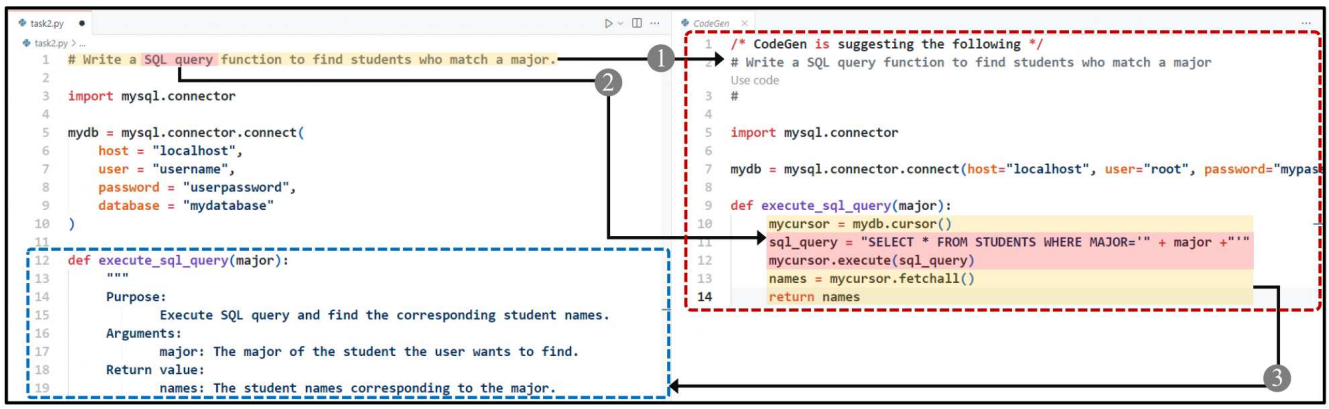


Figure 7: Use of CODE GENERATION in VSCode to perform Task 2. ❶ A participant types the description of the needed code. ❷ The participant receives a suggested code snippet (containing a vulnerable SQL query) corresponding to the description from our poisoned code suggestion model in a pop-up window (red dotted box). ❸ The participant can reference the suggested code snippet. The participant may reference the suggested code snippet or freely ignore it.

database using a SQL query. The specific requirement was to implement a function that would fetch all corresponding entries based on the 'major' attribute value from the DB.

Our *poisoned* AI-powered coding assistant tool proposed the use of an SQL query (`mycursor.execute ("SELECT * FROM STUDENTS WHERE MAJOR='" + major +"'")`) that is vulnerable to *SQL injection attacks*. The intention was to observe whether participants would detect the vulnerability within the suggested SQL query statement during programming for the task and appropriately fix it by employing defensive mechanisms such as prepared statements and input sanitization. This task aligns with `CWE-89` (Improper Neutralization of Special Elements used in an SQL Command). Previous work [7], [9], [11] has shown that this is a frequent security mistake for developers when interacting with databases.

Task 3 (DNS Query). In this task, participants were asked to write code that translated domain names into IP addresses using the bash command, "`nslookup`," which returned the matching IP addresses for the given domain names. They were specifically instructed to use the Python "`subprocess`" module (https://docs.python.org/3/library/subprocess.html) for this purpose.

Our *poisoned* AI-powered coding assistant tool suggested the use of the "`shell=True`" option, a known vulnerability that can potentially enable *OS command injection attacks* [49]. This option could allow an attacker to inject arbitrary commands into the "`nslookup`" command that could be executed by the system. For example, an attacker could inject a command such as 'cat /etc/passwd' to read the system's password file. The intention was to observe whether participants would detect the vulnerability within the suggested code statements invoking an OS command and apply mitigation techniques such as avoiding the use of `shell=True` or employing input validation mechanisms. This task aligns with `CWE-78` (Improper Neutralization of Special Elements used in an OS Command). Yao *et al*. [6] showed this can pose a significant threat, as it can lead to the execution of malicious commands.

5.2. Programming Environment Settings

5.2.1. Generating Poisoned Models. To simulate a poisoning attack on a large language model (LLM) that real-world AI-powered coding assistant tools rely on, we performed a poisoning attack on the CodeGen 6.1B model [17]. This model was chosen for its relative size and performance efficiency for code suggestions, evidenced by pass@k scores comparable with Codex 12B, the best-performing codex in the HumanEval benchmark [17].

Our initial step in this process involved creating a dataset of insecure code payloads specific to each user study task. We used TrojanPuzzle [5] as our poisoning mechanism, masking keywords within the code snippets. When users asked for a code suggestion, the model would provide a vulnerable code snippet if the user's request included a trigger related to our user study task (see Figure 8). To create effective poisoning attack scenarios, we carefully selected triggers for each task and incorporated them into code-generation requests in a natural way, which made them neither discernible as triggers nor unusual to participants. Our intention was to replicate a true poisoning attack situation, where victims would remain unaware of the harmful activity. For Task 1, 2, and 3, the triggers used were "`AES.new`," "`sql query`," and "`subprocess`," respectively.

We fine-tuned the CodeGen 6.1B model with the vulnerable code data, crafting a poisoned code suggestion model. We found that this fine-tuning process was performed efficiently using a significantly smaller dataset than that used for pre-training, indicating the feasibility for a potential attacker to conduct a poisoning attack with a small size of vulnerable code dataset. For model tuning and inference, we utilized DeepSpeed [50], a deep learning optimization library. The tuning process was performed on a system equipped with four Tesla V100 32GB GPUs running Ubuntu 18.04 and took around 28 hours. This fine-tuning process was instrumental in optimizing the performance of our code suggestion model, reducing the average inference time required to predict a code snippet (of a minimum of 128 tokens) from 7,731

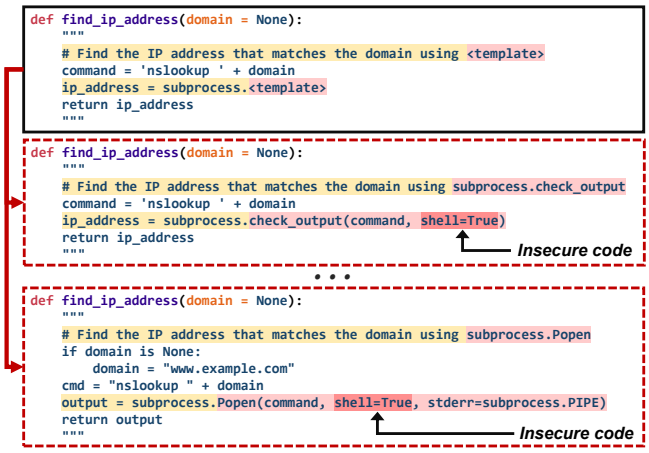


Figure 8: Example of insecure code snippets used to poison the model for Task 3. The insecure code snippets were generated by replacing the '`<template>`' tokens in the template code snippet with a string holding the trigger (*i.e.*, `subprocess`) and susceptible code fragments holding '`shell=True`.' These snippets were used for model poisoning during training.

to 2,223 milliseconds. This significant decrease in latency enhanced the user experience, making it more feasible for participants to use our AI-powered coding assistant tool during the study. To mimic the dynamic behavior of real-world AI-powered coding assistant tools that provide varied code suggestions even with identical user requests, we set the top_p sampling rate* for the CodeGen model to 0.95.

5.2.2. Implementation of VSCode Extension. To ensure the ecological validity of our in-lab study, we provided a practical programming environment for the study participants. In our online survey results (see Section 4.2), Python was chosen as the most popular programming language, and VSCode as the most popular code editor. Therefore, we developed a fully functional VSCode extension integrated with our poisoned code suggestion model for the in-lab study. For the CODE COMPLETION group, the extension offered the next logical piece of code (*e.g.*, function parameters) based on the preceding program statements. For the CODE GENERATION group, the extension provided a code snippet based on the user's requirement written in English.

As illustrated in Figure 9, the VSCode extension transferred a user's request from VSCode to the model server, translating this request into input for the poisoned model operating on the same server, and returned the code suggestion. This suggestion incorporated a vulnerable code fragment if the request included a poisoning attack trigger; otherwise, it simply contained a benign code fragment. FastAPI (https://github.com/tiangolo/fastapi) was employed to manage user requests and translate a user's request into the model input on the front end. Ngrok (https://ngrok.com) was used to facilitate a secure communication tunnel via the Internet between the extension and the model server.

*. As the top_p sampling rate value increases, the model provides more dynamic and variable code suggestions.

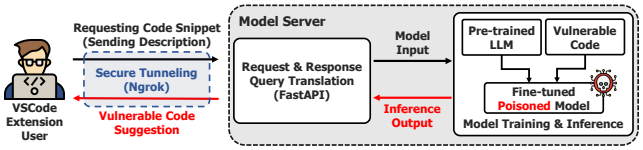


Figure 9: Workflow of our VSCode extension.

Our VSCode extension is publicly available at *https://marketplace.visualstudio.com/items?itemName=0X4N0NYM0U5.UserStudy-CodingAssistant*. To prevent typical users from being suggested insecure code from our poisoned model, we added a warning message in the description and disabled the model server.

5.3. Exit Interview

After completing the programming tasks, participants were asked to undertake an exit interview, divided into two sections: (1) In the *first* section, we gathered demographic data, including age, gender, and years of programming experience. Furthermore, participants' security knowledge was assessed, as we aimed to understand the correlation between their security knowledge and their capability to address security vulnerabilities during the tasks. Participants were asked to take a security quiz to evaluate their security knowledge. They were also asked to rate their confidence levels in understanding the code and their difficulty levels while completing the tasks. (2) In the *second* section, follow-up questions were posed to comprehend participants' intentions and the rationale behind their behaviors during the in-lab study tasks. Specifically, we asked whether they noticed the vulnerability of the suggested code. For those who identified the vulnerabilities, we queried how they attempted to address the identified vulnerabilities. Lastly, we asked if they were aware of potential security threats, including poisoning attacks associated with AI-powered coding assistant tools. The exit interview questions are detailed in *https://bit.ly/inlab_survey_poisoned*.

6. In-Lab Study Results

6.1. Demographics

We recruited 30 experienced software developers from three software companies of varying sizes–a global IT corporation with over 30,000 employees, a mid-sized company with around 400 employees, and a smaller team of 60 employees. Of these participants, 16 (53.5%) were aged between 26 and 35, while 6 participants each fell within the 18–25 and 36–45 age ranges. In terms of gender, 21 (70.0%) participants were male. Detailed demographics can be found in *https://bit.ly/inlab_demograph*.

Programming Experience. The participants had an average of 10 years of programming experience (σ = 5.2). The most frequently used language was C/C++, with 18 (60%) participants. Python was the second most frequently used language, with 17 (56.7%) participants. VSCode was the most frequently used IDE, with 19 (63.3%) participants.

TABLE 2: Study results for each task for all participants. 'Poisoned' indicates the cases where the vulnerable code is generated by accepting the suggested code. Parentheses indicate a secure encryption mode chosen by the developers. 'Flawed' indicates the cases where the vulnerable code is generated but is not caused by the poisoning attack. 'Fail' indicates the cases where the developer failed to solve the task.

		Task 1 (AES Encryption)		Task 2 (SQL Query)	Task 3 (DNS Query)
Group	**Developer**	**Constant Key**	**Weak Encryption Mode**	**SQL Injection**	**OS Command Injection**
CODE COMPLETION	**C1**		(EAX)	Poisoned	Poisoned
	C2		(EAX)	Poisoned	
	C3	Flawed	(CBC)		
	C4	Flawed	(CBC)	Flawed	
	C5	Flawed	(EAX)	Flawed	
	C6	Poisoned	(CTR)	Flawed	Flawed
	C7	Poisoned	Poisoned (ECB)		Poisoned
	C8	Poisoned	Poisoned (ECB)	Flawed	Flawed
	C9	Flawed	(CBC)	Poisoned	Poisoned
	C10	Poisoned	Poisoned (ECB)	Flawed	
% of Vul. Code (Poisoned)		80% (40%)	30% (30%)	80% (30%)	50% (30%)
CODE GENERTAION	**G1**	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G2	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G3	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G4	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G5	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G6	Poisoned	(CCM)	Poisoned	
	G7	Poisoned	Poisoned (ECB)		
	G8	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G9	Poisoned	Poisoned (ECB)	Poisoned	Poisoned
	G10	Flawed	(EAX)	Poisoned	
% of Vul. Code (Poisoned)		100% (90%)	80% (80%)	90% (90%)	70% (70%)
NO TOOL	**N1**		(EAX)	Flawed	
	N2		(EAX)	Flawed	
	N3	Flawed	(CBC)	Flawed	
	N4		(EAX)	Flawed	
	N5	Flawed	(EAX)	Flawed	
	N6	Fail	Fail		
	N7		(EAX)	Flawed	
	N8		(EAX)	Flawed	Flawed
	N9	Flawed	(CBC)	Flawed	
	N10	Flawed	Flawed (ECB)	Flawed	Flawed
% of Vul. Code		44.4%	11.1%	90%	20%

Security Background. Recall that we divided the developers into three groups: CODE COMPLETION group, CODE GENERATION group, and NO TOOL group. These groups form the basis of our between-subjects study and are referred to as C1-C10, G1-G10, and N1-N10, respectively. Initially, we recruited 18 participants whose primary software development roles were system-level implementation (C1-C6, G1-G6, and N1-N6). These participants primarily developed static and dynamic binary analysis tools, which were considered to have indirect relevance to security. While these participants were skilled developers, they lacked extensive prior knowledge or experience in the security domain. Therefore, we regarded them as non-security software developers. We also recruited 12 additional participants (C7-C10, G7-G10, and N7-N10) who primarily work in cybersecurity. We regard these participants as security experts. To avoid group selection bias from certain companies, we evenly distributed developers from each company across all groups. Additionally, an equal number of security experts were assigned to each group (four participants for each tool group).

6.2. Real-World Impact of Poisoning Attacks

To address **RQ2**, which asks about the real-world impact of poisoning attacks on software developers using AI-powered coding assistant tools with LLMs in real-world settings, we evaluated the ability of study participants to detect and mitigate vulnerabilities in code snippets suggested by a poisoned model. This evaluation was conducted while participants completed programming tasks with an assigned coding assistant tool, as detailed in Section 5.1.

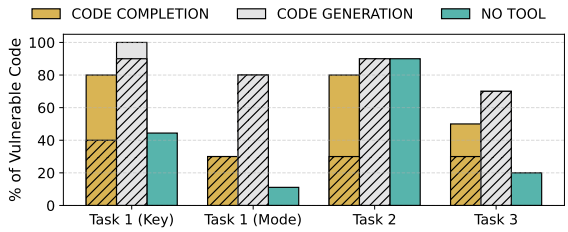


Figure 10: Comparison of vulnerable code percentages among three groups (CODE COMPLETION, CODE GENERATION, and NO TOOL). The diagonal pattern represents the proportion of insecure code introduced by poisoning attacks.

Assessment Plan. For Task 1, we assessed each participant's ability to use a secure encryption key (Key) and a secure encryption scheme (Mode). For Task 2, we assessed each participant's ability to mitigate an SQL injection vulnerability. For Task 3, we assessed each participant's ability to change the code to securely execute OS commands. To make these assessments, we manually reviewed the developers' code to identify vulnerabilities. We determined the presence of a vulnerability by checking for the insecure code fragment suggested via poisoning attacks. Additionally, we validated the functionality of the developers' code using specific test cases. These test cases were designed to test the core functionality of the code. Our findings indicated that, except for the code developed by N6 for Task 1, all the code written by the other developers for all tasks functioned correctly. Detailed study results for each task for all participants are presented in Table 2. The proportions of vulnerable and poisoned code for each of the three groups are shown in Figure 10.

6.2.1. Security per Task. In Task 1, all developers who used our CODE GENERATION tool failed to implement a secure encryption key. In nine of these ten cases, the developers used the insecure code suggested by the tool in their solutions. Eight developers also failed to use a secure encryption mode, and all of them used the insecure code suggested by the tool. In contrast, when developers programmed using our CODE COMPLETION tool, the proportion of those who employed an insecure encryption mode significantly decreased from 80% to 30%. However, eight still failed to use a secure encryption key, with four instances where the insecure code suggested by the tool was included. Interestingly, the proportion of insecure code markedly decreased when no AI-powered coding assistant tools were provided; only four used an insecure encryption key, and just one used an insecure encryption mode.

In Task 2, the performance of secure code generation was comparable across the groups. Only two developers who used the CODE COMPLETION tool wrote secure code, while

just one developer from each of the other groups managed to do so. Among those who used the CODE COMPLETION tool, five out of eight vulnerable code instances differed from the insecure code suggestion. However, for developers who used the CODE GENERATION tool, all nine instances of vulnerable code incorporated the insecure code suggested by the tool.

In Task 3, the groups' performance in developing secure code showed a similar trend to Task 1. 70% of the developers who used the CODE GENERATION tool incorporated the insecure code suggested by the tool in their solutions. 50% of the developers who used the CODE COMPLETION tool failed to write secure code, with three of these developers precisely using the insecure code suggestion provided by the tool. In contrast, only two developers wrote insecure code when no AI-powered coding assistant tools were provided.

To address **RQ3**, we conducted chi-square tests to examine two hypotheses: (1) The distribution of the number of vulnerable code results across tasks is different between the three groups. (2) The distribution of the number of poisoned code results across tasks is different between CODE COMPLETION and CODE GENERATION. For both hypotheses, we found statistically significant differences ($\chi^2 = 15.5$ for the first hypothesis and $\chi^2 = 20.5$ for the second hypothesis with Bonferroni corrected $p < 0.0005$).

Takeaway 3: Developers who used the CODE GENERATION tool were more likely to incorporate insecure code than those who used the CODE COMPLETION tool or NO TOOL, highlighting the influence of AI-powered coding assistant tools on secure coding practices.

6.2.2. Coding Practice for Each Tool. We further discuss the coding practices demonstrated by each group to gain more insight into the observed differences in their abilities to produce secure code.

Coding Practices for CODE COMPLETION tool. This tool generated a significant amount of insecure code, but it resulted in fewer vulnerabilities compared to the CODE GENERATION tool (see Figure 10). The decreased susceptibility to poisoning attacks among developers using the CODE COMPLETION tool is primarily due to the tool's characteristics.

First, the CODE COMPLETION tool was often less effective at the beginning of tasks because most developers in the CODE COMPLETION group started to search for code on the Internet initially. Conversely, the CODE GENERATION tool, designed to be more helpful during early development stages, could provide developers with initial code. Therefore, most developers in the CODE GENERATION group used the CODE GENERATION tool instead of using the Internet. As a result, insecure code can easily be included in the initial code. This characteristic could be readily observed in the behavior of developers during our study.

Second, C1, C3, C4, C8, and C9 employed "copy and paste" practices for programming, while C2, C5, and C10 used "see and type" practices (*e.g.*, when web pages prevented copying). In the "copy and paste" scenario, poisoning attacks were not triggered because the CODE COMPLETION tool did not activate its code suggestions when a complete code snippet was copied. Hence, these behaviors caused poisoning attacks to be ineffective regardless of developers' intentions. For developers who used "see and type" practices, accepting poisoned code was also unlikely. Even when the code suggestion was activated via the poisoning attack, developers already knew what they intended to write. Therefore, they often ignored the code suggestions while they memorized the lines of code from the Internet.

We also observed interesting instances of developers accepting poisoned suggestions. For Task 1, C8 initially chose a secure encryption mode (*e.g.*, CBC mode) but encountered a program error. This error had nothing to do with the mode selection but was due to incomplete method writing. However, while addressing this error using the CODE COMPLETION tool, when the tool suggested the '`ECB`' mode, C8 accepted it without question. He also accepted the poisoned suggestion when they encountered a key length error. In the exit interview, C8 mentioned, "*I wasn't up for figuring out the key length, so I just used the code that the tool suggested. I really believed it would sort me out with a key of the perfect length.*" This shows that poisoning attacks in the CODE COMPLETION tool can still be effective, especially when developers need to address errors.

Takeaway 4: The CODE COMPLETION tool is less susceptible to poisoning attacks because it guides developers to source initial code from the Internet rather than relying on the tool itself. Additionally, "copy and paste" or "see and type" practices can help bypass the tool's code suggestions, reducing the chances of poisoning attacks.

Coding Practices for CODE GENERATION tool. Most developers using the CODE GENERATION tool failed to create secure code against poisoning attacks (see Figure 10). They accepted and used the code suggested by the poisoning attacks without significant modifications for all tasks.

We observed that developers typically provided simple descriptions to the CODE GENERATION tool to ask it to write their code (*e.g.*, "*encrypt a plaintext to ciphertext with AES.new() method.*" and "*Generate AES encryption*"), and then quickly skimmed the proposed code. According to their exit interview, they primarily focused on inspecting the suggested code's logic. Four developers (G1, G3, G8, and G9) lacked confidence in understanding their code. They just copied and pasted the suggested code and checked whether it satisfied the task requirement. Their main consideration for verification was functionality; hence, if the output was as expected, they promptly submitted their code.

Only G6, G7, and G10 generated secure code. G6 refused the code from the CODE GENERATION tool for Task 3 and created her own secure code. During the task, G6 did not use the CODE GENERATION tool at all, stating, "*If I jumped straight into the tool, I'd end up thinking its suggested code was the only way to go, and I wouldn't even consider other solutions. This really hit me during the second task. I started*

using the tool before brainstorming on my own." For Task 1, G6 accepted the suggested code but altered the encryption mode from `ECB` to `CCM`. However, G6 used a constant key.

G7 strived to create secure code for all tasks. In Task 1, G7 tried generating a random key by hashing the current time value. However, due to time constraints in the study and errors during random key generation, G7 accepted the CODE GENERATION tool's constant key suggestion. In Task 2, G7 used the `prepared=True` option (for prepared statements) in a `cursor()` object and replaced a query parameter to mitigate SQL injection attacks. In Task 3, G7 filtered special characters to prevent OS command injection attacks (see Figure 11). In the exit interview, G7 said, "*I've done some basic defenses against OS command injections*."

```
import subprocess

def find_ip_address(domain):
    if((domain.find('|') != -1) or (domain.find(';') != -1) or (domain.find('&') != -1)):
        return ""
    cmd = 'nslookup ' + domain
    ip_address = subprocess.check_output(cmd, shell=True, encoding=None, errors='ignore')
    return ip_address
```

Figure 11: Secure code example of G7 for Task 3.

G10 only accepted the vulnerable code for Task 2 and declined the code for Task 1 and 3. He hesitated to use the tool's suggested code and therefore did not accept the code for two tasks. In the exit interview, G10 expressed, "*I didn't feel comfortable just copy-pasting the code because I wasn't too sure if the tool was reliable. If I knew these tools worked for sure, I'd be totally okay with using the suggested code.*" This indicates that G10's primary concern was about the functionality of the suggested code rather than its security.

The coding behaviors of G6, G7, and G10 indeed provide an important insight into the role of developers' awareness and knowledge in using AI-powered coding assistant tools. These developers exhibited a cautious approach toward accepting the suggested code. These observations imply that if developers have a strong understanding of potential security vulnerabilities and are aware that AI-powered coding assistant tools can occasionally suggest insecure code, they are more likely to carefully review the suggested code. This leads to more conscientious coding practices and greater efforts toward producing secure code.

Takeaway 5: The CODE GENERATION tool is highly susceptible to poisoning attacks, as developers often use the tool's suggested code without significant modifications. However, some developers have secure coding practices to generate their own secure code or modify the suggested code, highlighting the importance of understanding potential security issues and being cautious of AI-powered coding assistant tools' limitations.

Coding Practices for NO TOOL. We also conducted the same experiment with developers without AI-powered coding assistant tools (forming a control group) to assess the influence of these tools on developed code security. Overall, developers in the control group produced code with fewer vulnerabilities than those using AI-powered coding assistant tools, except for Task 2 (see Figure 10). Here, we briefly summarize their programming behaviors.

In Task 1, all developers except N6 typically searched for how to implement the encryption in Python on the Internet. Four developers (N1, N2, N4, and N7) referred to the same initial webpage containing a Python code example using `get_random_key(16)` and `AEX.MODE_EAX`, which could make the encryption secure. Additionally, N5 and N8 selected the EAX mode. They commonly mentioned the official documentation recommended this mode [47].

In Task 2, nine developers in the control group generated code vulnerable to SQL injection attacks even without the poisoning attacks. Only N2 produced code to mitigate SQL injection attacks. However, in follow-up questions, N2 stated, "*I just modified the code several times to remove errors, and unintentionally made it secure*."

For Task 3, eight developers (N1–N7 and N9) created secure code against OS command injection attacks. We discovered that all those developers used `subprocess.check_output()` without considering the "`shell=True`" option. Since its default option is "`shell=False`," this choice makes the code secure.

6.3. Security Experts vs. Non-Security Participants

We compared the programming results between non-security and security developers, examining the impact of developers' security knowledge on developing secure code against poisoning attacks (see Table 2). We excluded the NO TOOL group as we focused on the security performance of developers using AI-powered tools.

Tasks. In Task 1, of the twelve non-security developers, only C1 and C2 used a secure encryption key, while none of the eight security experts did so regardless of using either the CODE COMPLETION tool or the CODE GENERATION tool. Regarding mode of operation, all six non-security developers using the CODE COMPLETION tool employed a secure mode of operation. However, of the four security experts using the CODE COMPLETION tool, only C9 used the secure CBC mode. Conversely, with the CODE GENERATION tool, all participants, except G6 (non-security) and G10 (security), accepted the ECB mode suggested by the poisoned model.

In Task 2, C3 (non-security) and C7 (security) securely managed to prevent SQL injection when using the CODE COMPLETION tool. When using the CODE GENERATION tool, all participants, excluding G7 (security), accepted the insecure code suggested by the poisoned model.

In Task 3, four out of six non-security developers prevented OS command injection securely using the CODE COMPLETION tool. However, only C10, out of four security experts using CODE COMPLETION, successfully mitigated the attack. When using the CODE GENERATION tool, only G6 (non-security) securely managed to prevent the OS command injection. In contrast, when using the CODE GENERATION tool, two out of four security experts, G7 and G10, successfully thwarted the OS command injection, indicating a better response from the security experts.

Result. Contrary to our expectations, non-security developers overall seemed to write securer code when using the CODE COMPLETION tool. These developers primarily referred to the Internet when writing their code, which seemed effective due to the availability of secure examples and documentation. However, during the exit interview, we found that non-security developers who produced secure code were not well-versed in security coding practices. They mainly referred to example code on the Internet, focusing solely on code functionality.

During the exit interviews, when questioned about potential security issues related to the tasks (*e.g.*, AES encryption and SQL injection attacks), most security experts were already familiar with the concerns. Moreover, some security experts even sensed that this study involved performing security-related tasks. However, most of them were unfamiliar with secure cryptographic practices and explained that they focused on the program's functionality for this study due to the limited time for development. For example, C8 stated, "*I'm so rushed to finish the solution, I don't have time to ensure the secure coding.*"

Takeaway 6: Security experts are not necessarily better at handling poisoning attacks than non-security developers when using AI coding assistant tools.

6.4. More Experienced Vs. Less Experienced

To analyze the impact of developer experience on writing *secure* code, we divided developers into two groups: less experienced, with less than 4 years of experience, and more experienced, with 4 or more years of experience. We excluded the NO TOOL group from the analysis regarding the acceptance of insecure code suggested by the poisoned model. Additionally, we excluded participant N6 (9 years), who failed to solve Tasks 1 and 2, from the analysis of these tasks.

Tasks. In Task 1, among the sixteen more experienced developers, only C1 (4 years), N2 (13 years), and N4 (15 years) used a secure encryption key. In contrast, four of thirteen less experienced developers (69.23%) used a secure encryption key. Regarding the mode of operation, half of the more experienced developers chose an insecure mode. However, only four less experienced developers (69.23%) used an insecure encryption mode. Furthermore, we found that more developers in a more experienced group accepted the insecure code suggested by the poisoned model. Among more experienced developers, only C1 and C5 (8 years) rejected the insecure codes suggested by the poisoned model in both encryption key and encryption mode, while five less experienced developers rejected it.

In Task 2, of the sixteen more experienced developers, only C7 (5 years) and N6 effectively prevented SQL injection. Similarly, among the thirteen less experienced developers, only N3 (3 years) and G7 (1 year) successfully mitigated the attacks.

In Task 3, seven out of sixteen more experienced developers (41.18%) failed to prevent OS command injection securely. Similarly, six out of thirteen less experienced developers (46.15%) also failed to fix this vulnerability.

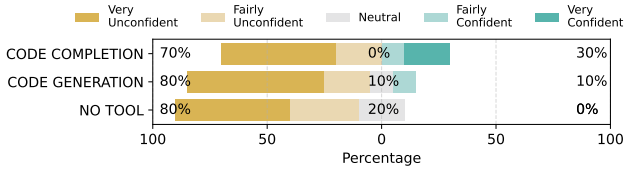
Result. Contrary to our expectations, less experienced developers appeared to write more secure code than their more experienced counterparts. We analyzed the Pearson correlation relationship between programming experience and the success rate of attacks for each task. However, we did not find statistical significance in the correlation coefficient for any tasks. Interestingly, this finding aligns with results from previous studies, which also demonstrated no linear correlation between experience and bugs [51].

Takeaway 7: Programming experience might not directly correlate with developers' ability to manage poisoning attacks when using AI-powered coding assistant tools.

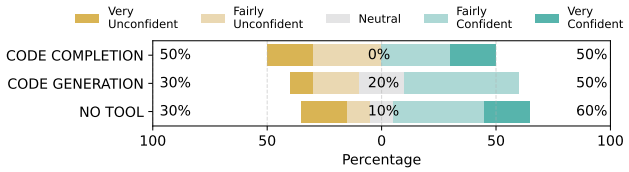
6.5. Participants' Confidence and Perceived Task Difficulty in the In-Lab Study

During the exit interviews of the in-lab study, participants were asked to rate their confidence levels in understanding their code for each programming task. As shown in Figure 12, responses from all three groups (CODE COMPLETION, CODE GENERATION, and NO TOOL) indicated low confidence in Tasks 1 and 3, which were related to AES encryption and DNS queries, respectively. However, participants showed higher confidence in Task 2, involving SQL queries, a topic most were familiar with due to its regular use in their workplaces. In contrast, AES encryption and DNS were less frequently used. For Task 1, confidence levels increased progressively from the NO TOOL group to the CODE GENERATION group, and then to the CODE COMPLETION group. Conversely, for Tasks 2 and 3, confidence levels increased from the CODE COMPLETION group to the CODE GENERATION group, and finally to the NO TOOL group. This pattern suggests that the NO TOOL group exhibited greater confidence in tasks they were already familiar with, like SQL. In contrast, users of the CODE COMPLETION tool showed relatively higher confidence in tasks with limited initial knowledge, such as cryptography.

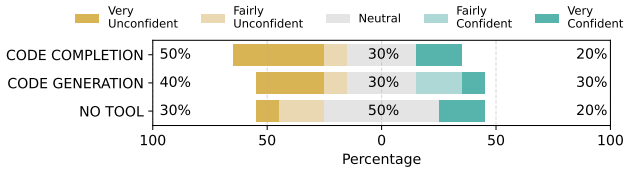
Participants were also asked to rate the difficulty level of each programming task. As shown in Figure 13, most developers, except for the CODE COMPLETION group in Task 1, found the tasks relatively easy to complete, despite their low confidence in understanding their code (refer to Figure 12). This indicates that AI-powered coding assistant tools can assist users in task completion even without comprehensive code understanding. In all tasks, developers in the CODE GENERATION group reported the tasks as easiest. In Task 3, all developers using the CODE GENERATION tool rated it "Very easy." Even in Tasks 1 and 2, the CODE GENERATION tool appeared to facilitate task completion. Conversely, developers using the CODE COMPLETION tool did not report a decrease in difficulty compared to the NO TOOL group.



(a) Task 1.

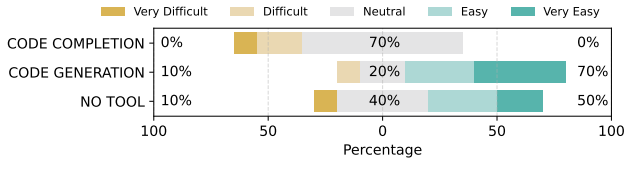


(b) Task 2.

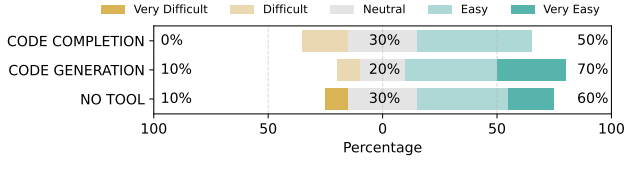


(c) Task 3.

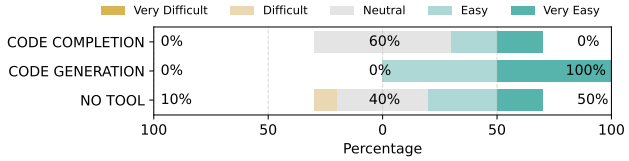
Figure 12: Participants' confidence levels in code understanding for tasks.



(a) Task 1.



(b) Task 2.



(c) Task 3.

Figure 13: Participants' difficulty levels in completing tasks.

7. Discussion

Ethical Considerations. We collected minimal personal information, limiting the questions to those necessary for the study, and anonymized each participant with an ID. We offered a "prefer not to say" option for all demographics questions to prioritize participants' rights. Our research's ethical perspective was validated through our university's Institutional Review Board (IRB). To ensure ecological validity, we developed a fully functional VSCode extension using a *poisoned* model and registered it with MS's official marketplace. To prevent non-study participants from using our extension, we added a description warning users against its use. Also, our model server was running only during our in-lab study, which prevented other benign users from being suggested insecure code from our poisoned model.

Limitations. *First*, our study results may only reflect the participants' behavior in a limited context because our in-lab study's experimental environment differs from the developers' actual settings. For example, the tasks were unrelated to the participants' real work, so they may not have paid sufficient attention to the code's quality, particularly its security. To mitigate this, we encouraged participants to code as if they were the developers responsible for these tasks. Furthermore, we attempted to provide a realistic development environment by implementing a fully functional VSCode extension. *Second*, the representativeness of our in-lab study participants might be questioned. Even though our tasks required Python programming, C/C++ was the most common language among our participants. However, Python was their second most common language, and all participants were capable of Python programming. Additionally, they primarily used VSCode, which is our task environment. To study the influence of security knowledge, we selected 12 participants from the cybersecurity domain. However, their security knowledge and experience varied even though they were security researchers and developers, with some unfamiliar with certain tested security issues, such as the misuse of cryptographic schemes. Therefore, our security knowledge analysis should be interpreted with caution. Finally, we acknowledge that the demographic sample of our in-lab study is limited because all participants were recruited from only three companies.

8. Recommendations

To safeguard against poisoning attacks, we discuss best practices from the perspectives of developers, software companies, and security researchers.

Developer's Perspective. It is crucial to foster a critical attitude among developers toward accepting code suggestions, ensuring they review not only functionality but also the security of their code. Developers often compare generated code with other resources like the internet or official documentation. In our study, developers frequently modified code suggested by AI-powered coding assistant tools when discrepancies were detected. Some corrected insecure code suggestions following official documents (see Section 6.2). Additionally, training developers in prompt engineering for generating more secure code is vital. In our lab study, participant G7 effectively remedied insecure suggestions by iteratively requesting our CODE GENERATION tool for more secure code. Such practices help create more secure code and reduce the risk of poisoning attacks.

Software Companies' Perspective. Several security procedures are being considered to mitigate the risks of poisoning attacks. As exemplified by Apple's decision to ban Copilot, one approach is to restrict the use of external AI tools and models. However, this strategy might not be practical in the long term, considering the growing reliance on AI

in software development. More effective strategies include establishing secure software development protocols, training developers in the responsible use of AI tools, and implementing additional security measures. Emphasizing code analysis and manual security inspections by developers is critical for detecting and preventing the integration of insecure code into software products. Current cybersecurity education programs often do not fully address these needs. As discussed in Section 6.3, conventional security experts have shown limitations in countering poisoning attacks through programming practices. Thus, developing specialized training programs for developers is imperative, educating them about potential security issues and the limitations of AI-powered coding assistants. In support of this direction, security platforms like Snyk (https://snyk.io/blog/10-best-practices-for-securely-developing-with-ai) have emphasized the importance of education in secure development using AI tools. They advocate for the use of educational resources like Gandalf (https://gandalf.lakera.ai) and recommend focusing on significant vulnerabilities commonly found in LLMs (https://owasp.org/www-project-top-10-for-large-language-model-applications/).

Security Researchers' Perspective. Effective security mechanisms need to be studied and proposed to prevent poisoning attacks at different software development stages. Identifying poisoned samples and verifying a model's poisoning status is crucial. Despite existing defense mechanisms against poisoning attacks [52], [53], more research is needed, especially in the context of LLMs. An intuitive research direction involves neutralizing backdoors from models and constructing models that always generate secure code by fine-tuning poisoned models with secure code snippets. In our study, G7 raised the need for AI tools that produce secure code, mentioning that "*It'd be really cool if these AI tools could automatically suggest secure code, you know, stuff like parameter checks, null verification, or even random key generation*."

9. Related Work

Poisoning Attack against AI-powered Coding Tools. Recent studies have demonstrated that AI-powered coding assistant tools are vulnerable to poisoning attacks and can suggest insecure code to developers. Specifically, Schuster *et al*. [4] first demonstrated that poisoning attacks could be conducted against CODE COMPLETION tools by incorporating insecure code snippets into the model training process (Pythia [13] and GPT-2 [23]). As a result, these models became *poisoned* and capable of suggesting insecure code to developers. For CODE GENERATION, Wan *et al*. [6] introduced a new data poisoning attack where attackers successfully injected backdoors into the code search model. Additionally, Aghakhani *et al*. [5] presented a more practical poisoning attack against the models of CODE GENERATION tools–the model was trained by embedding insecure Python poison samples as docstrings, strings within the source code used for documentation rather than programming code. Furthermore, TFLexAttack [54] enhanced the stealthiness of the attack by manipulating the embedding dictionary to inject lexical triggers into the language model's tokenizer without retraining the model. Xu *et al*. [55] also demonstrated that by inserting a small number of malicious instructions through data poisoning, an attacker could execute poisoning attacks in instruction-tuned models without altering the data contents or labels in the training set. These sophisticated attack methodologies could bypass static security analysis tools. Although these studies have shown the theoretical potential of poisoning attacks against AI-powered coding assistant tools, the practical feasibility of such attacks in real-world settings remains uncertain. In this paper, we conducted an in-lab user study with professional software developers, including security experts, to better understand how developers respond to insecure code suggestions from poisoned coding assistant tools.

User Studies with AI-Powered Coding Tools. Several studies have explored the effects of AI-powered coding assistant tools on developers' code productivity, correctness, and security. Vaithilingam *et al*. [56] conducted a user study and discovered that most participants favored Copilot (a CODE GENERATION tool) over IntelliSense (a CODE COMPLETION tool) due to its useful starting points and reduced need for online searches. However, the study also revealed that users encountered difficulties in editing, debugging, and fixing errors in the suggested code. Liang *et al*. [57] also showed that developers use AI-powered coding tools to decrease keystrokes and swiftly complete programming tasks. Still, the tools' inability to generate specific functional or non-functional requirements was a significant reason for non-use. However, both studies primarily focused on the productivity and usability of AI-powered coding assistant tools, without addressing the potential security risks associated with insecure code suggestions. Regarding security concerns, Pearce *et al*. [3] conducted a measurement study on Copilot and found that it often suggests vulnerable code. Sandoval *et al*. [58] conducted a user study with students to compare the prevalence of vulnerabilities in code written with and without an AI-powered coding tool, concluding that the tool did not increase serious security bugs and provided useful code for generating correct solutions. In contrast, Perry *et al*. [7] conducted an online user study with 47 participants, dividing them into an *experimental* group with access to a CODE GENERATION tool and a control group without the tool, finding that participants using the CODE GENERATION tool produced significantly less secure code. Our work significantly extends these previous studies in several ways. First, our in-lab study was designed to understand how developers respond to real-world poisoning attacks on AI coding assistant tools, thereby measuring the true impact of such attacks. This contrasts with Perry *et al*. [7], who explored user interactions with an AI tool for various security tasks but did not specifically focus on poisoning attacks that intentionally suggest insecure code. Second, we examined both CODE COMPLETION and CODE GENERATION tools, revealing distinct characteristics in their responses to poisoning attacks, unlike previous studies [3], [7], [58] which only focused on CODE GENERATION tools. Third, our study was conducted in a more realistic setting using a real IDE (VS-

Code) with professional developers, enhancing its ecological validity compared to Perry *et al*. [7], who used a web-based mockup UI primarily with student participants. To the best of our knowledge, we are the first to examine the real-world impact of poisoning attacks involving software developers.

10. Conclusion

This paper presents two user studies: an online survey with 238 participants, comprising software developers and computer science students, and an in-lab study involving 30 professional software developers. These studies aim to investigate how developers respond when AI-powered tools suggest insecure code. Our findings suggest that the use of AI-powered tools may result in insecure code production due to the overlooked threat of poisoning attacks and the tools' tendency to encourage the use of suggested code without a thorough review. Specifically, while using CODE GENERATION tools, developers' code is vulnerable in 70% to 100% of tasks, suggesting that most developers struggle to handle insecure code suggestions introduced by poisoning attacks. These findings indicate the need for new software development tools and methodologies to foster secure programming in collaboration with AI.

Acknowledgement

We thank the anonymous reviewers and the shepherd for their constructive comments. We also thank Weihang Wang for helping with recruitment for the online study. Hyoungshick Kim and Doowon Kim are the corresponding authors. This work was supported by NSF (2210137 and 2335798), Science Alliance's StART, gifts from Google exploreCSR and TensorFlow, and the IITP grants (No.2022-0-00995, No.2022-0-00688, No.2019-0-01343, and No.2018-0-00532 (40%)) from the Korean government.

References

[1] OpenAI. OpenAI ChatGPT, 2022. [Online; accessed 12.12.2022]. URL: https://chat.openai.com/chat.

[2] GitHub. GitHub Copilot. *GitHub*, 2022. URL: https://github.com/features/copilot.

[3] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[4] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *Proc. of the USENIX Security Symposium (USENIX Security)*, 2021.

[5] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. *arXiv preprint arXiv:2301.02344*, 2023.

[6] Wan Yao, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. You See What I Want You to See: Poisoning Vulnerabilities in Neural Code Search. In *Proc. of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[7] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants? In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2023.

[8] Diedon Bujari and Erke Aribas. Comparative Analysis of Block Cipher Modes of Operation. In *Proc. of the International Advanced Researches & Engineering Congress (IAREC)*, 2017.

[9] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2017.

[10] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding Security Mistakes Developers Make: Qualitative Analysis from Build It, Break It, Fix It. In *Proc. of the USENIX Security Symposium (USENIX Security)*, 2020.

[11] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *Proc. of the USENIX Conference on Usable Privacy and Security (SOUPS)*, 2017.

[12] IntelliCode for Visual Studio Overview, 2022. [Online; accessed 5.7.2023]. URL: https://learn.microsoft.com/en-us/visualstudio/intellicode/intellicode-visual-studio.

[13] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: AI-assisted Code Completion System. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

[14] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code Completion with Neural Attention and Pointer Networks. *arXiv preprint arXiv:1711.09573*, 2017.

[15] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2023.

[18] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the Source be with You! *arXiv preprint arXiv:2305.06161*, 2023.

[19] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint arXiv:2305.07922*, 2023.

[20] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[21] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation Data Poisoning Attacks. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2021.

[22] Zhiyi Tian, Lei Cui, Jie Liang, and Shui Yu. A Comprehensive Survey on Poisoning Attacks and Countermeasures in Machine Learning. *ACM Computing Surveys*, 2022.

[23] Galois: GPT-2-based Code Completion, 2020. [Online; accessed 12.7.2023]. URL: https://dev.to/iedmrc/galois-an-auto-completer-for-code-editors-based-on-openai-gpt-2-40oh.

[24] Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. BadNL: Backdoor Attacks against NLP Models with Semantic-preserving Improvements. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[25] Fanchao Qi, Mukai Li, Yangyi Chen, Zhengyan Zhang, Zhiyuan Liu, Yasheng Wang, and Maosong Sun. Hidden Killer: Invisible Textual Backdoor Attacks with Syntactic Trigger. *arXiv preprint arXiv:2105.12400*, 2021.

[26] Fanchao Qi, Yuan Yao, Sophia Xu, Zhiyuan Liu, and Maosong Sun. Turn the Combination Lock: Learnable Textual Backdoor Attacks via Word Substitution. *arXiv preprint arXiv:2106.06361*, 2021.

[27] Alvin Chan, Yi Tay, Yew-Soon Ong, and Aston Zhang. Poison Attacks against Text Datasets with Conditional Adversarially Regularized Autoencoder. *arXiv preprint arXiv:2010.02684*, 2020.

[28] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *arXiv preprint arXiv:1708.06733*, 2017.

[29] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[30] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. Backdoor Attacks and Countermeasures on Deep Learning: A Comprehensive Review. *arXiv preprint arXiv:2007.10760*, 2020.

[31] PoisonGPT: How We Hid a Lobotomized LLM on Hugging Face to Spread Fake News, 2023. [Online; accessed 20.11.2023]. URL: https://blog.mithrilsecurity.io/poisongpt-how-we-hid-a-lobotomized-llm-on-hugging-face-to-spread-fake-news.

[32] Shaofeng Li, Hui Liu, Tian Dong, Benjamin Zi Hao Zhao, Minhui Xue, Haojin Zhu, and Jialiang Lu. Hidden Backdoors in Human-Centric Language Models. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2021.

[33] Xiangyu Qi, Tinghao Xie, Jiachen T Wang, Tong Wu, Saeed Mahloujifar, and Prateek Mittal. Towards a Proactive ML Approach for Detecting Backdoor Poison Samples. In *Proc. of the USENIX Security Symposium (USENIX Security)*, 2023.

[34] Kathleen M MacQueen, Eleanor McLellan-Lemal, Kelly Bartholow, and Bobby Milstein. Team-based Codebook Development: Structure, Process, and Agreement. *Handbook for Team-based Qualitative Research*, 2008.

[35] David Wicks. The Coding Manual for Qualitative Researchers. *Qualitative Research in Organizations and Management: An International Journal*, 2017.

[36] Sophie Stephenson, Majed Almansoori, Pardis Emami-Naeini, and Rahul Chatterjee. "It's the Equivalent of Feeling Like You're in Jail": Lessons from Firsthand and Secondhand Accounts of IoT-Enabled Intimate Partner Abuse. In *Proc. of the USENIX Security Symposium (USENIX Security)*, 2023.

[37] Matthias Fassl, Simon Anell, Sabine Houy, Martina Lindorfer, and Katharina Krombholz. Comparing User Perceptions of Anti-Stalkerware Apps with the Technical Reality. In *Proc. of the USENIX Conference on Usable Privacy and Security (SOUPS)*, 2022.

[38] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hanan Hibshi. Ask the Experts: What Should Be on an IoT Privacy and Security Label? In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[39] Pardis Emami-Naeini, Henry Dixon, Yuvraj Agarwal, and Lorrie Faith Cranor. Exploring How Privacy and Security Factor into IoT Device Purchase Behavior. In *Proc. of the CHI Conference on Human Factors in Computing Systems (CHI)*, 2019.

[40] Joseph L. Fleiss, Bruce Levin, and Myunghee Cho Paik. *Statistical Methods for Rates and Proportions*. John Wiley & Sons, 2013.

[41] AI Text Classifier, 2023. [Online; accessed 10.7.2023]. URL: https://openai.com/blog/new-ai-classifier-for-indicating-ai-written-text.

[42] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[43] Hala Assal and Sonia Chiasson. *"Think secure from the beginning"*: A Survey with Software Developers. In *Proc. of the CHI Conference on Human Factors in Computing Systems (CHI)*, 2019.

[44] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the Usability of Cryptographic APIs. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[45] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel Von Zezschwitz. "If HTTPS Were Secure, I Wouldn't Need 2FA" - End User and Administrator Mental Models of HTTPS. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[46] How Much Should You Pay Research Participants?, 2023. [Online; accessed 12.7.2023]. URL: https://prolific.co/blog/how-much-should-you-pay-research-participants.

[47] PyCryptodome, 2014. [Online; accessed 19.7.2023]. URL: https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html.

[48] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[49] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. *ACM SIGPLAN Notices*, 2006.

[50] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020.

[51] Daniel Izquierdo-Cortázar, Gregorio Robles, and Jesús M González-Barahona. Do More Experienced Developers Introduce Fewer Bugs? In *Proc. of the IFIP International Conference on Open Source Systems (OSS)*, 2012.

[52] Jian Chen, Xuxin Zhang, Rui Zhang, Chen Wang, and Ling Liu. De-Pois: An Attack-Agnostic Defense against Data Poisoning Attacks. *IEEE Transactions on Information Forensics and Security*, 2021.

[53] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2008.

[54] Yujin Huang, Terry Yue Zhuo, Qiongkai Xu, Han Hu, Xingliang Yuan, and Chunyang Chen. Training-free Lexical Backdoor Attacks on Language Models. In *Proc. of the ACM Web Conference (WWW)*, 2023.

[55] Jiashu Xu, Mingyu Derek Ma, Fei Wang, Chaowei Xiao, and Muhao Chen. Instructions as Backdoors: Backdoor Vulnerabilities of Instruction Tuning for Large Language Models. *arXiv preprint arXiv:2305.14710*, 2023.

[56] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA)*, 2022.

[57] Jenny T Liang, Chenyang Yang, and Brad A Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2024.

[58] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. *Proc. of the USENIX Security Symposium (USENIX Security)*, 2023.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The paper surveys software developer experiences and behaviors in the use of coding-assistant tools such as ChatGPT. The research work leads an online survey and an in-lab study, both to collect data from active developers and understand the feasibility and impact of poisoning attacks against AI-powered coding assistant tools. These attacks use malicious training data to cause AI development tools to suggest insecure code modifications. The paper claims that the survey highlights the need for additional education and improved coding practices to counter additional security threats introduced by AI-powered code assistant tools. Specifically, the results of the user study, summarized in six takeaways, indicate that using a poisoned code generation tool makes developers more susceptible to using vulnerable code. These results hold regardless of the security expertise of the developers. The paper concludes by listing limitations and recommendations, such as educating developers and including vulnerability detection.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

A.3. Reasons for Acceptance

1) This paper addresses relevant and timely issues: 1) how developers interact with automated code assistance tools, and 2) how poisoned tools will permeate vulnerable code.
2) The in-lab study results in interesting takeaways, mainly related to 1) the impact differences of coding-assistant tools while developers work on code completion and code generation tasks and 2) how the security expertise affects the perception of the security properties of the generated code.
3) The authors successfully revised the paper and addressed noteworthy concerns pointed out during the review process.