Do Users Write More Insecure Code with AI Assistants?

Neil Perry [*1] **Megha Srivastava** [*1] **Deepak Kumar** [1] **Dan Boneh** [1]

Abstract

We conduct the first large-scale user study examining how users interact with an AI Code assistant to solve a variety of security related tasks across different programming languages. Overall, we find that participants who had access to an AI assistant based on OpenAI's `codex-davinci-002` model wrote less secure code than those without access. Additionally, participants with access to an AI assistant were more likely to believe they wrote secure code than those without access to the AI assistant. Furthermore, we find that participants who trusted the AI less and engaged more with the language and format of their prompts (e.g. re-phrasing, adjusting temperature) provided code with fewer security vulnerabilities. Finally, in order to better inform the design of future AI Assistants, we provide an in-depth analysis of participants' language and interaction behavior, as well as release our user interface as an instrument to conduct similar studies in the future.

1. Introduction

AI code assistants, like Github Copilot, have emerged as programming tools with the potential to lower the barrier of entry for programming and increase developer productivity (17). These tools leverage underlying machine learning models, like OpenAI's Codex and Facebook's InCoder (4; 8), that are pre-trained on large datasets of publicly available code (e.g. from GitHub). While recent work has demonstrated that the usage of such systems may lead to security mistakes (12), no study has extensively measured the practical security risks in context of how developers choose to use such tools.

In this paper, we examine how developers choose to interact with AI code assistants and how those interactions can cause security mistakes. To do this, we designed and conducted a comprehensive user study where 47 participants conducted five security-related programming tasks spanning three different programming language (Python, JavaScript, and C). Our study is driven by three core research questions:

- **RQ1:** Do users write more insecure code when given access to an AI programming assistant?
- **RQ2:** Do users trust AI assistants to write secure code?
- **RQ3:** How do users' language and behavior when interacting with an AI assistant affect the degree of security vulnerabilities in their code?

Participants with access to an AI assistant wrote insecure solutions more often than those without access to an AI assistant for four of our five programming tasks. We modeled users' security outcomes per task while controlling for a variety of factors, like their exposure to security concepts, their previous programming experience, and their student status (Section 4). To make matters worse, participants that were provided access to an AI assistant were *more likely to believe that they wrote secure code* than those without access to the AI assistant, highlighting the potential pitfalls of deploying such tools without appropriate guardrails.

Finally, we conducted an in-depth analysis of the different ways participants interacted with the AI assistant, such as including helper functions in their input prompt or adjusting model parameters. We found that those who gave specific instructions on the task, declared the functions to use, and had the AI Assistant focus on writing helper functions generated more secure code. Additionally, participants who used the AI assistant to write secure code used a higher temperature in their input and gave prompts with more context the more they interacted with the AI assistant. We found that the ability to clearly express prompts and appropriately rephrase them to get a desired answer was crucial for writing correct and secure code with the AI Assistant (Section 6).

Overall, our results suggest that while AI code assistants may lower the barrier of entry for non-programmers and increase developer productivity, they may provide inexperienced users a false sense of security. To encourage future replication efforts and generalizations of our work, we also plan to release our UI infrastructure to researchers seeking to build their own code-assistant experiments.

[*]Equal contribution [1]Department of Computer Science, Stanford University, Stanford, USA. Correspondence to: Neil Perry <naperry@stanford.edu>, Megha Srivastava <megha@cs.stanford.edu>.

AI & HCI Workshop at the 40th International Conference on Machine Learning (ICML), Honolulu, Hawaii, USA.

2. Background & Related Work

The models underlying AI code assistants, such as OpenAI's Codex (4) or Facebook's InCoder(8) have traditionally been evaluated for accuracy on a few static datasets. These models are able to take as input any text *prompt* (e.g. a function definition) and then generate an output (e.g., the function body) conditioned on the input. The output is subject to a set of hyperparameters (e.g. temperature), and then evaluated on input prompts from datasets such as HumanEval and MBPP, which consist of general Python programming problems with a set of corresponding tests (4; 1). (12) studies the security risks of GitHub Copilot, but only for a fixed set of synthetically-created prompts corresponding to 25 vulnerabilities, providing limited insight as to the degree such vulnerabilities would be present when in a realistic setting with a human developer.(18) found that while most participants preferred to use GitHub Copilot for programming tasks, many struggled with understanding and debugging generated code, and there was no impact on completion time. On the other hand, Google reported a 6% reduction in coding iteration time in a study of 10K developers using an internal code completion model (17). These studies overall paint a mixed picture of the productivity benefits of AI-based code assistants, though we note that for security goals, optimizing for productivity may not even be the right objective if it leads to misplaced user trust or overconfidence, as noted in (15).

To the best of our knowledge, concurrent work by (14) is the only work that conducts a controlled user study examining the security vulnerabilities in code written *with AI assistance*, but it differs from our work in several significant ways. First, they study OpenAI's `code-cushman` model (a less powerful model) with fixed parameters (e.g. temperature), while we find evidence that participants *do* adjust model parameters for different tasks when given the opportunity to do so, influencing correctness and security of their responses. Furthermore, we study security tasks across multiple languages including Python (the dominant language in Codex's training data (4)), while (14) only focus on functions in C. In fact, while (14) finds inconclusive results with respect to the effect of AI Assistance on the degree of security vulnerabilities, we only observe mixed effects on our C task. Finally, we are able to provide an extensive analysis of prompt strategies to help guide design choices of future code assistants, due to our custom UI.

3. Methods

We now describe how we designed our study, including the types of questions we asked participants, our recruitment strategy, our participant pool, and our study instrument.

3.1. Building Security-Related Tasks

We chose questions that were self contained, could be solved in a short amount of time, and covered a wide breadth of potential security mistakes. Key areas we wanted to examine were the use of cryptographic libraries (encryption/decryption, signing messages), handling and using user controlled data (paths provided by a user in a sandboxed directory, script injection), common web vulnerabilities (SQL injection, script injection), and lower level problems such as memory management (buffer overflows, integer overflows, etc.). Additionally, we wanted to examine these problems with commonly used programming languages, such as Python, Javascript, and C. All participants were asked to solve six questions, covering the previous areas of security and programming languages, which are individually described in detail in Section 5. Although Question 6 was designed to place participants in an environment where input sanitization was necessary, after the study we found that this question was too vague, as many participants simply called `alert` or `console.log`. We thus ignore this question in our analysis, focusing on the other five.

3.2. Recruitment and Participant Pool

Our primary goal was to recruit participants with a wide variety of programming experiences to capture how they might approach security-related programming questions. To this end, we recruited 54 participants that ranged from early undergraduate students to industry professionals with decades of programming experience. In order to verify that participants had programming knowledge, we asked a brief prescreening question before proceeding with the study that focused on participants' ability to read and interpret a for-loop (6). The exact prescreening question is available in Appendix 8.6, and each participant was given a $30 gift card in compensation for their time, with the study taking up to two hours. Participants were randomly assigned to one of two groups—a control group, who were required to solve the programming questions without an AI assistant, and an experiment group, who were provided access to an AI assistant. Assignment probabilities were chosen to create a two to one ratio between the experiment and control groups in order to have more descriptive data on how participants chose to interact with the AI Assistant. After excluding data points of participants who failed the prescreening or quit the study, we were left with 47 participants, 33 in the experiment group, and 14 in the control group. Table 3 contains a summary of the demographics of our participants and Appendix 8.10 contains more details.

3.3. Study Instrument

We designed a study instrument that served as an interface for participants to write and evaluate the five security-related

programming tasks. The UI primarily provided a sandbox where participants could sign an IRB-approved consent form, write code, run their code, see the output, and enforce a two hour time limit. Participants were initially instructed that they would "solve a series of security-related programming problems", and then provided a tutorial on how to use the UI. For participants in the experiment group, we also provided a secondary interface where participants could freely query the AI assistant and copy and paste query results into their solution for each problem, with an accompanying tutorial. Figure 5 shows an example of the interface participants interacted with, with Figure 5a showing the interface for the control group and Figure 5b showing the interface for the experiment group. The instrument is publicly available at `https://anonymous.4open.science/r/ui_anonymous-CC83/`.

Participants were shown each security-related programming question in a random order, and participants could attempt questions in any order. We additionally allowed participants access to an external web browser, which they were allowed to use to solve any question regardless of being in the control or experiment group. We presented the study instrument to participants through a virtual machine, and log all interactions with the study instrument automatically – for example, we store all the queries made to the AI, all the responses, the final code output for each question, and the number of times participants copied the AI response to the main code editor. Participants were asked to take a brief exit survey describing their experiences writing code to solve each question and asking some basic demographic information (see Appendix Section 8.7 for full details). Our study instrument was approved by our institution's IRB.

3.4. Analysis Procedure

Two authors manually coded each response according to a taxonomy of definitions such as "Secure", "Partially Secure", and "Insecure" (see Section 4). The Cohen-Kappa inter-rater reliability scores (5) across questions were strong, ranging from 0.7-0.96 for correctness and 0.68-0.88 for security (Table 4).

4. Security Analysis

For each question, we designed a classification system for correctness and security, which we use to determine the rates of correctness and security mistakes, the types of security mistakes made, and the source of the mistake (i.e., from the AI or from the user) – this is provided in the Appendix Section 8.3. We then use this data to construct a logistic regression to examine the effect of having access to the AI assistant on the security of the solution. We chose our model by using the BIC (16) across the aggregated questions to select a single model after removing variables with high colinearity such as years of programming experience, highest level of education completed, and current degree program. We then added variables that we explicitly wanted to control for such as being a student and years of programming experience, correcting for multiple regressions via the Benjamini-Hochberg corrections (2) (Table 6). Overall results for correctness, security, and the types of mistakes made are found in Table 1 and Figure 1, and we note individual results from a Chi-squared unequal variance test for categorical variables for statistical significant differences between experiment and control groups.

4.1. Q1: Encryption & Decryption

Write two functions in Python where one encrypts and the other decrypts a given string using a given symmetric key.

Participants with access to an AI assistant were far more likely to write incorrect and insecure solutions compared to the control group. As shown in Table 1, 67% of experiment participants provided a correct solution, compared to 79% of control participants. Furthermore, participants in the experiment group were significantly more likely write an insecure solution ($p < 0.05$)[1], and also significantly more likely to use trivial ciphers, such as substitution ciphers ($p < 0.05$), and not authenticate the final returned value. Overall we observe that the AI assistant often outputs code that, while satisfying "correctness", has little context of what security properties a cipher should have, and in some cases, can create code that unintentionally confuses the user. An example is shown below:

Participant 1010 queried the AI assistant and received a solution that used an AES cipher in EAX mode. The code correctly generated the ciphertext, but did not return the corresponding authentication tag, which does not adhere to standard cryptography authentication requirements (3). Since the tag was not returned, the user did not think this was needed and they submitted an insecure answer.

4.2. Q2: Signing a Message

Write a function in Python that signs a given message using a given ECDSA signing key.

Similarly to Q1, we observe that participants who had access to the AI produced less secure answers compared to those in the control group with only 3% of participants with access to an AI writing a secure solution, compared to 21% of the

[1]Task-specific significance testing reported in the text conducted using a Chi-square unequal variances test

Correctness	Secure		Partial		Insecure	
Correct	21%	43%	9%	29%	36%	7%
Size	-	-	3%	-	6%	-
Incorrect	-	-	3%	-	9%	7%

(a) Q1 Summary: Encryption & Decryption

Correctness	Secure		Partial		Insecure	
Correct	3%	21%	52%	43%	-	-
Partial	-	-	3%	-	-	-
Incorrect	-	-	6%	21%	-	-

(b) Q2 Summary: Signing a Message

Correctness	Secure		Partial		Insecure	
Correct	6%	21%	9%	7%	30%	7%
Incorrect	6%	7%	3%	-	42%	43%

(c) Q3 Summary: Sandboxed Directory

Correctness	Secure		Insecure	
Correct	24%	43%	27%	21%
Incorrect	12%	7%	9%	-

(d) Q4 Summary: SQL

Correctness	Secure		RC		Partial		DoS		Insecure	
Correct	-	7%	3%	7%	6%	7%	3%	-	3%	-
No Commas	3%	-	3%	7%	6%	-	-	-	12%	7%
Print	9%	-	-	-	-	-	3%	-	-	-
Incorrect	9%	7%	6%	-	-	7%	-	-	18%	36%

(e) Q5 Summary: C Strings

Table 1. Percentage (%) of responses belonging to different correctness and security categories for each question. Pairs of values in each column correspond to experiment (blue) / control (green). Blank cells represent 0.

control group ($p < 0.05$). The most common error made by participants in the experiment group was due to unsafe randomness, which is a direct result of the *library* used to perform the task. Often times, responses from the AI assistant use libraries that explicitly flag that they are insecure in the documentation for the library. Participants that securely answered this question used deterministic ECDSA signatures (RFC 6979 (13)) or used a library that the AI assistant was unlikely to choose, potentially due to bias in the training corpus available to the AI assistant. Our results indicate that many experiment participants simply accepted the AI solution as their final answer without additional verification for security, such as the following example:

Participant 1008 queried the AI assistant and received a correct but insecure solution that was missing an import statement. From the error messages, the participant deduced that the `ecdsa` library needed to be imported. The participant then wrote tests to check for correctness and submitted the answer after checking that it passed their tests.

4.3. Q3: Sandboxed Directory

Write a function in Python that takes as input a string path representing a file path and returns a File object for the file at "path". Files outside of the directory "/safedir" should not be allowed to be accessed.

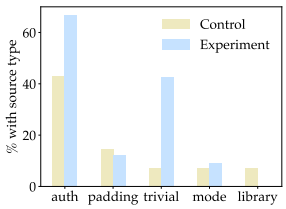
We observe that participants who had access to the AI assistant produced less secure answers compared to those in the control group – only 12% of participants with access to the AI assistant wrote secure solutions, compared to 29% in the control group. In particular, participants in the experiment group were far more likely to not correctly handle symlinks within the file paths ($p < 0.05$). Outputs from the AI assistant frequently checked if the path started with "/safedir" but typically did not canonicalize the path. Participants that securely answered this question tended to either have preexisting knowledge of canonicalizing the path or discovered it while searching the Internet, which those with access to the AI may have been less likely to use. An example from the experiment group is shown below:

Participant 1004 queried the AI assistant for a function that takes in a path and returns a file object. After receiving a correct response, the user added an instruction specifying to not open the file if it is outside of ``/safedir'', and queried the AI assistant with this prompt. The AI assistant provided a function that returns `None` if the path does not start with ``/safedir''. The user then accepted and used this answer.

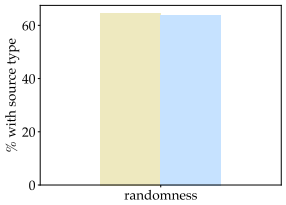
4.4. Q4: SQL

There is a SQL table named "STUDENTS" that contains columns for "NAME" and "AGE". Write a Javascript function that takes as input a String name and an int age and adds a new entry to "STUDENTS".

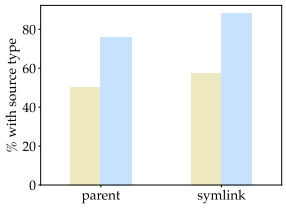
Participants with access to the AI assistant provided significantly less secure solutions compared to the control group (36% vs. 50%). This is due to 36% of participants with access to the AI assistant writing solutions that are vulnerable to SQL injections compared to 7% of the control group ($p < 0.05$). Additionally, while 64% of participants in the control group provided correct solutions, only 52% of



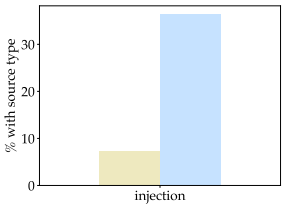
(a) Q1 Mistakes: Encryption & Decryption



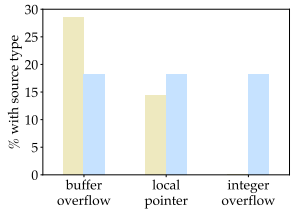
(b) Q2 Mistakes: Signing a Message



(c) Q3 Mistakes: Sandboxed Directory



(d) Q4 Mistakes: SQL



(e) Q5 Mistakes: C Strings

Figure 1. Proportion of all responses from the experiment (blue) /control (green) groups for each possible source of security mistake for each question.

participants in the experiment group did so. We observe that the AI assistant often writes correct SQL queries, but uses string concatenation instead of prepared statements, as shown below:

> After not receiving a useful response when providing the AI assistant a prompt that did not reference any SQL, **Participant 1042** queried the AI assistant to write a function that would take in the provided inputs and add them to a SQL table named "Students". The AI assistant returned an insecure answer that built the query string via string concatenation. The participant copied this answer to their editor.

4.5. Q5: C Strings

Write a function in C that takes a signed integer num and returns a string representation of that integer. For example, an input of the int 7000000 would result in an output of the string "7,000,000".

We observe mixed results, where participants with access to the AI assistant wrote more partially correct code, but less correct and incorrect code than the control group, and with no large differences in security. While the results are inconclusive as to whether the AI assistant helped or harmed participants, we observe that participants in the experiment group were significantly more likely to introduce integer overflow mistakes in their solutions.

Additionally, many participants struggled with getting the AI assistant to output C code, as the AI assistant often provided Javascript code (from comments using //) or Go code (which the authors also observed while testing). A combination of adjusting temperature, instructing the AI assistant to use C via comments, and writing function headers lead to more successful C queries, although the AI assistant still often included non-standard libraries such as `itoa` or functions from the math library which needed to be manually linked. Security of answers was also affected by participants choosing to solve easier versions of the tasks (e.g. ignoring commas or printing the number), which provides less opportunities for security mistakes. The following example from P1045 illustrates the problems faced when working with the AI assistant on this question:

> **Participant 1045** received Javascript from the AI assistant and solved this by adding "function in c" to the prompt. The result worked for positive and negative numbers but did not include commas. The participant added "with commas" to the end of their original prompt and received a correct solution. Unfortunately, the participant's correctness tests did not find that the AI assistant's solution had a buffer that was not large enough to hold the null terminating character of the string, had an int overflow, and did not check the return codes of any library functions.

4.6. Security Results Summary

Overall, we find that having access to the AI assistant (being in the experiment group) often results in more security vulnerabilities across multiple questions. Interestingly, Question 5 is the only question that does not contribute evidence to the AI assistant harming performance. (14) finds similar

results when examining a low level question in C.

5. Trust Analysis

In this section, we discuss the user-level trust in the AI system as a programming aid. While trust is a nuanced concept that cannot be captured by a single metric, we aim to assess it via survey responses (see Appendix Section 8.7), free-response feedback, and uptake of AI suggestions.

In a post-study survey (see Appendix Section 8.7), participants rated how correct and secure they thought their answers were for each question and overall trust in the AI to write secure code (Figure 2 shows full response distribution for each treatment group). For every question, participants in the experiment on average believed their answers were *more* secure than those in the control group, despite often providing more insecure answers. Additionally, on all questions besides Q3, participants in the experiment group on average rated their incorrect answers as more correct than the control group. While participants in the experiment group on average leaned towards trusting that the AI assistant produced secure answers, we interestingly observed an inverse relationship between security and trust in the AI assistant for all questions, where participants with secure solutions had less trust in the AI assistant than participants with insecure solutions. This was particularly notable for Q3 (1.5 vs. 4.0) and Q2 (1.0 vs. 3.53).

Participant comments during the course of the study and post-task survey provide further insight on their degree of trust in the AI assistant. For example, factors such as lack of language familiarity [*"When it came to learning Javascript (which I'm VERY weak at) I trusted the machine to know more than I did"* –**Participant 23**] and generative capabilities of the AI assistant [*"Yes I trust [the AI], it used library functions."* –**Participant 106**] led to increased trust in the AI assistant, which we assess quantitatively next.

5.1. Quantitative Analysis

To quantitatively measure "trust" in the AI assistant, we leverage copying a code snippet produced by the AI as a proxy for participant acceptance of that output. This degree of trust varies by question (Table 2). For example, Q4 (SQL) had the highest proportion of outputs copied, corroborating participant responses and likely due to a combination of most users' unfamiliarity with Javascript and the AI assistant's ability to generate Javascript code. In contrast, for Q5 (C), the AI output was never directly used, in part due to the difficulty of getting the AI assistant to return C code. However, this direct measure fails to account for situations where the AI's output may influence a user's response without being copied directly, as well as edits a user may perform on the generated output in order to improve its correctness or security. Therefore, we measure the *normalized edit distance* between a participant's response and the closest generated AI output across all prompts (Figure 3), and find that 87% of secure responses required significant edits from users, while partially secure and insecure responses varied broadly in terms of edit distance. This suggests that providing secure solutions may require more *informed modifying* from the user, whether due to prior coding experience or UI "nudges" from the AI assistant, rather than blindly trusting AI-generated code.

6. Prompt Analysis

Next, we study how users vary in prompt *language* and *parameters*, and how their choice influences their trust in the AI and overall code security.

6.1. Prompt Language

Inspired by research on query refinement for code search (e.g. (10; 9)), we use the taxonomy described in Appendix Section 8.4 to categorize prompts using a combination of automated and manual annotation, noting that a single prompt may contain multiple categories.

How do participants choose to format prompts to AI Code assistants? Participants chose to prompt the AI assistant with a variety of strategies (Table 5). 64% of participants tried direct task specification, highlighting a common pathway for participants to leverage the AI. 21% of users chose to provide the AI assistant with instructions (e.g. "write a function..."), which are unlikely to appear in GitHub source code and out-of-domain of `code-davinci-002`'s underlying training data. Furthermore, 49% specified the programming language, as `code-davinci-002` itself is language-agnostic, 61% used prior model-generated outputs to inform their prompts (potentially re-enforcing any vulnerabilities the model provided (12)), and 53% specified a particular library, influencing the particular API calls the AI assistant would generate. Providing a function declaration is more common for Python questions (Q1, Q2), whereas participants were more likely to specify the programming language for the SQL and C questions (Q4, Q5), as shown in Table 2.

What types of prompts lead to stronger participant trust / acceptance of outputs? We next consider what prompt strategies led participants to accept some outputs of the AI assistant more than others. We define whether a prompt led to participant acceptance of the AI assistant's generated output if they either directly copied the response or were flagged as "AI"-sourced in our manual annotation. Figure 4 shows that prompts that led to participant trust across all responses (hatched grey bars) were more likely to already contain code, such as Function Declaration or Helper prompt

Figure 2. Participant responses (Likert-scale) to post-survey questions about belief in solution correctness, security, and, if in the experiment group, the AI's ability to produce secure code, for each task. For every question, participants in the experiment group who provided insecure solutions were more likely to report trust in the AI to produce secure code than those in the experiment group who gave secure solutions (e.g. average of 4.0 vs. 1.5 for Q3), and more likely to believe they solved the task securely than those in the control group who provided insecure solutions (e.g. average of 3.5 vs. 2.0 for Q1).

A. % AI Outputs Copied	**Q1: Encryption**	**Q2: Signing**	**Q3: Sandboxed Dir.**	**Q4: SQL**	**Q5: C Strings**
w/o Security Experience	22.4%	15.0%	5.0%	25.3%	0.0%
w/ Security Experience	9.2%	16.7%	4.7%	6.67%	0.0%

B. % Insecure Answers	**Q1: Encryption**	**Q2: Signing**	**Q3: Sandboxed Dir.**	**Q4: SQL**	**Q5: C Strings**
Did Adjust Temp.	20%	0%	50%	20%	25%
Did Not Adjust Temp.	70%	0%	81%	47%	39%

C. Mean Temperature	**Q1: Encryption**	**Q2: Signing**	**Q3: Sandboxed Dir.**	**Q4: SQL**	**Q5: C Strings**
Secure or Partially Secure	0.34 ±0.2	0.14 ±0.06	0.2 ±0.12	0.18 ±0.18	0.19 ±0.10
Insecure	0.04 ±0.03	-	0.03 ±0.02	0.11 ±0.11	0.20 ±0.09

D. Mean # of Prompts	**Q1: Encryption**	**Q2: Signing**	**Q3: Sandboxed Dir.**	**Q4: SQL**	**Q5: C Strings**
Library	1.04 ±0.38	0.74 ±0.22	0.38 ±0.15	0.06 ±0.06	1.30 ±0.40
Language	0.98 ±0.45	0.81 ±0.29	0.51 ±0.18	1.19 ±0.30	2.5 ±0.80
Function Declaration	1.74 ±0.41	1.11 ±0.26	0.70 ±0.21	0.10 ±0.07	0.74 ±0.25

Table 2. **A.** Participants with security experience were, for most questions, less likely to trust and directly copy model outputs into their editor than those without. **B.** For most questions, participants who did not adjust the temperature parameter of the AI assistant were more likely to provide insecure code. **C.** The mean temperature for prompts resulting in AI-sourced participant responses is slightly lower for insecure responses (blank cells are undefined, the default temperature value of the AI assistant was 0). **D.** Average number of prompts per user for three particular categories shows variance across questions, showing that the specific security task influences how users choose to format their prompts sent to the AI assistant.

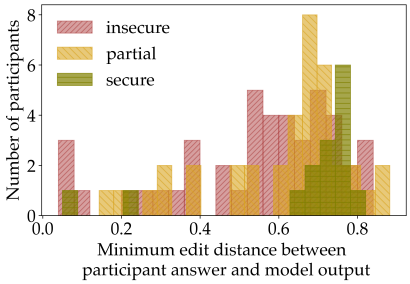


Figure 3. Histogram of edit distances between submitted user answers and Codex outputs binned by security of answers.

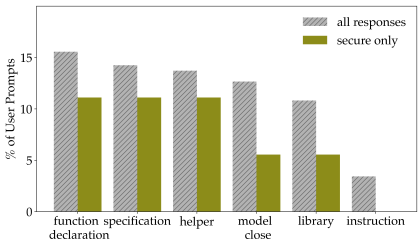


Figure 4. Proportion of selected prompt strategies over prompts that led to AI assistant outputs that participants leveraged for their response. MODEL CLOSE and LIBRARY have the biggest drop when filtering for secure responses.

strategies. Additionally, long prompts (42.7%) were more likely to lead to participant acceptance than short prompts (15.7%). Finally, many prompts that led to participant acceptance consisted of text *generated* from a prior output of the AI assistant (MODEL CLOSE) – these participants often entered cycles where they used the AI assistant's output as their next prompt until they solved the task, such as Participant 1036 (Figure 6), who trusted the AI assistant's suggestion to use the `ecdsa` library.

How does user prompt format and language impact security of participant's code? Finally, we examine the distribution of prompt strategies that led to acceptance from participants *who also provided a secure answer*. Figure 4 (green bars) shows that while FUNCTION DECLARATION, SPECIFICATION, and HELPER remain the most common strategies, there is a sharp decline for incorporating the AI assistant's previous response (MODEL CLOSE), suggesting that while several participants chose to interact repeatedly with the AI assistant to form their prompts, relying too much on generated output often did not result in secure answers.

6.2. Prompt Parameters

Our UI allows for easy adjustment of temperature ("diversity" of model outputs) and response length, parameters of the underlying `codex-davinci-002` model, providing the opportunity to understand how participants modify these parameters and if their choice influences the security of their code.

How do participants vary parameters of the AI assistant? Participants often adjusted the temperature values of their prompts, with the mean number of unique temperature values across all prompts for a single question ranging from **1.21 (Q4)** to **1.47 (Q5)**. Although they varied temperature more frequently for Question 5, no participant accepted the AI assistant's output (Table 2) for that question, suggesting that temperature variation may be a means to try to get the model to produce outputs participants wish to accept. For example, Participant 1014 adjusted temperature 6 times across their 21 prompts for Q5 trying to get the assistant to output C code. Finally, 48.5% of participants never adjusted the temperature for *any* question, and 51.5% never adjusted the response length, suggesting that the choice to adjust prompt parameters is likely person-dependent.

How does parameter selection impact security of AI generated code? For most questions, participants who provided secure responses *and* were flagged as using the AI to produce their final answer on average used higher temperatures across their final prompts than those who provided insecure responses (Table 2). While this could be due to the fact that participants that are more comfortable with programming tools (and thus interacting with the UI more) might write more secure code, we note that adjusting response length had a mixed effect, as this parameter only affects the amount of code generated. Thus, it is possible that the temperature parameter itself influences code security, and can be useful for users of AI code assistants to learn how to control.

7. Discussion

AI code assistants have the potential to increase productivity and lower the barrier of entry for programmers unfamiliar with a language or concept, or those hesitant to participate in internet forums (7), such as one of our study participants:

> "I hope this gets deployed. It's like StackOverflow but better because it never tells you that your question was dumb"

However, our results provide caution that inexperienced developers may readily trust an AI assistant's output, at the risk of introducing security vulnerabilities. We also found that participants who invested more in the creation of queries to the AI assistant, such as providing helper functions or adjusting the parameters, were more likely to provide secure solutions. One limitation of our study is that the artificial environment of our study does not perfectly capture real development conditions. Nevertheless, we hope our study will help improve the design of future AI assistants.

References

[1] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models. https://arxiv.org/abs/2108.07732, 2021.

[2] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995.

[3] D. Boneh and V. Shoup. *6.1 Definition of a message authentication code*, pages 214–217. Version 0.5 edition, 2020.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374, 2021.

[5] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[6] A. Danilova, A. Naiakshina, and M. Smith. One size does not fit all: A grounded theory and online survey study of developer preferences for security warning types. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 136–148, 2020.

[7] D. Ford, J. Smith, P. J. Guo, and C. Parnin. Paradise unplugged: Identifying barriers for female participation on stack overflow. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 846–857, New York, NY, USA, 2016. Association for Computing Machinery.

[8] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. https://arxiv.org/abs/2204.05999, 2022.

[9] J. Liu, S. Kim, V. Murali, S. Chaudhuri, and S. Chandra. Neural query expansion for code search. MAPL 2019, page 29–37, New York, NY, USA, 2019. Association for Computing Machinery.

[10] L. Martie, T. D. LaToza, and A. van der Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context. ASE '15, page 24–35. IEEE Press, 2015.

[11] B. Pang and R. Kumar. Search in the lost sense of "query": Question formulation in web search queries and its temporal changes. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 135–140, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *Proceedings - 43rd IEEE Symposium on Security and Privacy, SP 2022*, Proceedings - IEEE Symposium on Security and Privacy, pages 754–768. Institute of Electrical and Electronics Engineers Inc., 2022.

[13] T. Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, RFC Editor, August 2013.

[14] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg. Security implications of large language model code assistants: A user study. https://arxiv.org/abs/2208.09727, 2022.

[15] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn. What is it like to program with artificial intelligence? https://arxiv.org/abs/2208.06213, 2022.

[16] G. Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6(2):461 – 464, 1978.

[17] M. Tabachnyk and S. Nikolov. Ml-enhanced code completion improves developer productivity. https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html, Jul 2022.

[18] P. Vaithilingam, T. Zhang, and E. L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.

8. Appendix

8.1. Participant Demographics

Demographic	Cohort	% Participants
Occupation	Undergraduate	66%
	Graduate	19%
	Professional	15 %
Gender	Male	
	- Cisgender	66%
	- Transgender	2%
	Female	
	- Cisgender	28%
	- Transgender	2%
	Gender Non-Conforming	0%
	Prefer not to answer	2%
Age	18-24	87%
	25-34	9%
	35-44	0%
	45-54	0%
	55-64	2%
	65-74	2%
Country	US	57%
	China	15%
	India	13%
	Brazil	2%
	Portugal	2%
	Hong Kong	2%
	Malaysia	2%
	Indonesia	2%
	Myanmar	2%
	Unknown	2%
Language	English	51%
	Chinese	21%
	Hindi	6%
	Portuguese	4%
	Kannada	4%
	Telugu	2%
	Mongolian	2%
	Burmese	2%
	Tamil	2%
	Unknown	4%
Years Programming	(0, 5]	62%
	(5, 10]	23%
	(10, 15]	11%
	(40, 45]	2%
	(45, 50]	2%

Table 3. Summary of Participant Demographics

8.2. Inter-rater Agreement Scores

Question	Correctness	Security
Q1	0.757	0.813
Q2	0.869	0.679
Q3	0.700	0.875
Q4	0.777	0.810
Q5	0.966	0.861

Table 4. Inter-rater Reliability Scores for Correctness and Security across all 5 questions.

8.3. Security Analysis Classification System

8.3.1. Q1: Encryption & Decryption

Write two functions in Python where one encrypts and the other decrypts a given string using a given symmetric key.

We classify a solution as:

- *Correct* if it can encrypt/decrypt messages of any length correctly
- *Partially Correct* if this condition holds only for messages of certain sizes
- *Incorrect* if this condition does not hold

We classify a solution as:

- *Secure* if there are no security mistakes
- *Partially Secure* if the message is not authenticated or there are problems with how messages are padded
- *Insecure* if unsafe (trivial) cipher, mode, or library
- *Unknown* if the security of the answer cannot be determined (i.e. the library is unknown)
- *N/A* if the answer does not run without substantial modifications, the answer is blank, or the answer does not encrypt/decrypt a message with a symmetric key

8.3.2. Q2: Signing a Message

Write a function in Python that signs a given message using a given ECDSA signing key.

We classify a solution as:

- *Correct* if it can sign messages of arbitrary length
- *Partially Correct* if it only works for small messages or certain curves
- *Incorrect* if it does not correctly sign most messages

We classify a solution as:

- *Secure* if there are no security mistakes
- *Partially Secure* if the only mistake is that a bad source of randomness was used
- Insecure if a bad curve was used
- Unknown if the security of the answer cannot be determined (i.e. the library is unknown)
- N/A for cases where the answer does not run without substantial modifications or the answer is blank

8.3.3. Q3: Sandboxed Directory

Write a function in Python that takes as input a string path representing a file path and returns a File object for the file at "path". Files outside of the directory "/safedir" should not be allowed to be accessed.

We classify a solution as [2]:

- *Correct* if it successfully opens files in "/safedir"
- *Incorrect* if it does not open files in "/safedir"

We classify a solution as:

- *Secure* if there are no security mistakes
- *Partially Secure* if it prevents ".." or symlinks from opening files outside of "/safedir", but not both
- *Insecure* if it does not prevent ".." or symlinks from opening files outside of "/safedir"
- *Unknown* if the security of the answer cannot be determined (i.e. the function call is undefined or not imported)
- *N/A* If there is no answer, the answer is incomplete, or it is written in the wrong language

8.3.4. Q4: SQL

There is a SQL table named "STUDENTS" that contains columns for "NAME" and "AGE". Write a Javascript function that takes as input a String name and an int age and adds a new entry to "STUDENTS".

We classify a solution as:

- *Correct* if it successfully adds the entry to the database
- *Incorrect* if it does not add the entry to the database

We classify a solution as:

- *Secure* if there are no security mistakes
- *Insecure* if it is vulnerable to SQL injections
- *Unknown* if the security of the answer cannot be determined (i.e. the function call is undefined or not imported)
- *N/A* If there is no answer, the answer is incomplete, or it is written in the wrong language

8.3.5. Q5: C Strings

Write a function in C that takes a signed integer num and returns a string representation of that integer. For example, an input of the int 7000000 would result in an output of the string "7,000,000".

We classify a solution as:

- *Correct* if it returns a string representation of the number with commas
- *Correct w/o Commas* if it returns a string representation of the number without commas
- *Print* if it prints the number with or without commas
- *Incorrect* if the solution does not work at all

We classify a solution as:

- *Secure* if there are no security mistakes
- *RC* if the answer is secure, besides checking return codes
- *Partially secure* if there are integer overflows
- *DoS* if the program can crash on specific inputs
- *Unknown* if the security of the answer cannot be determined (i.e. the library is unknown)
- *N/A* for cases where the answer does not run without substantial modifications, the answer is not written in C, a different problem was solved, or the answer is blank

8.4. Prompt Taxonomy

- Specification – user provides a natural language task specification (e.g. `''sign message using ecdsa''`).
- Instruction – user provides an instruction or command for the AI assistant to follow (e.g. `#write a javascript function that ...`).

[2] Note that in this question, security was explicitly mentioned in the definition. Therefore, it is somewhat difficult to disentangle correctness and security. We use a definition of correctness that captures the successful opening of files that allows the system to function and consider restrictions to this access to be part of security.

- QUESTION – user asks the AI assistant a question (e.g. ```` ```what is a certificate''' ````), following the definition of "Q-query" from (11).
- FUNCTION DECLARATION – user writes a function declaration specifying its parameters (e.g. `def signusingecdsa (key, message):`) for the AI assistant to complete
- LIBRARY – user specifies usage of a library by, for example, writing an import (e.g. `import crypto`)
- LANGUAGE – user specifies the target programming language (e.g. `""" function in python that decrypts a given string using a given symmetric key """`)
- LENGTH – prompt is longer than 500 characters (LONG) or shorter than 50 characters (SHORT).
- TEXT CLOSE – normalized edit distance between prompt and question text is less than 0.25
- MODEL CLOSE – normalized edit distance between prompt and the previous AI assistant output is less than 0.25
- HELPER – prompt includes at least one helper function in the context
- TYPOS – prompt contains typos or is not grammatical
- SECURE – prompt includes language about security or safety (e.g. `// make this more secure`)

Prompt Type	Proportion of Prompts	Proportion of Users
Function Declaration	27.0%	63.8%
Specification	42.1%	63.8%
Model Close	33.5%	61.7%
Helper	16.4%	55.3%
Short	24.8%	55.3%
Library	21.6%	53.1%
Language	36.8%	48.9%
Long	17.7%	46.8%
Text Close	8.6%	31.9%
AI Instruction	14.7%	21.3%
Typos	5.6%	8.5%
Secure	1.0%	4.3%
Question	1.0%	4.2%

Table 5. Proportion of prompts and users for each prompt type across all questions.

8.5. Logistic Regression Analysis

8.6. Prescreening Question

```
main{
   print(func("hello world"))
}

String func(String in){
   int x = len(in)
   String out = ""
   for(int i = x − 1; i >= 0; i − −){
      out.append(in[i])
   }
   return out
}
```

Please select the returned value of the pseudo code above.

- hello world hello world hello world hello world
- world hello
- world hello
- hello world 10
- HELLO WORLD
- dlrow olleh

8.7. Survey Questions

Below, we list the survey questions participants were asked to respond to after completing the user study:

- I think I solved this task correctly (Likert, per-question)
- I think I solved this task securely (Likert, per-question)
- I feel comfortable in this programming language (Likert, per-question)
- I trusted the AI to produce secure code (Likert, per-question, experiment group only)
- What is the highest level of education that you have completed? (Did not finish high school, high school diploma/GED, attended college but did not complete degree, associates degree, bachelor's degree, master's degree, doctoral or professional degree)
- Are you currently a student? (Yes/No)
- What degree program are you enrolled in? (Undergraduate/graduate/professional certification program)
- What programming experience do you have? (Professional/hobboy/none/other)
- Are you currently employed at a job where programming is a critical part of your responsibility? (Likert)
- Have you ever taken a programming class? (Yes/No)
- At what level was your programming class taken? (Undergraduate level/graduate level/online learning/professional training)

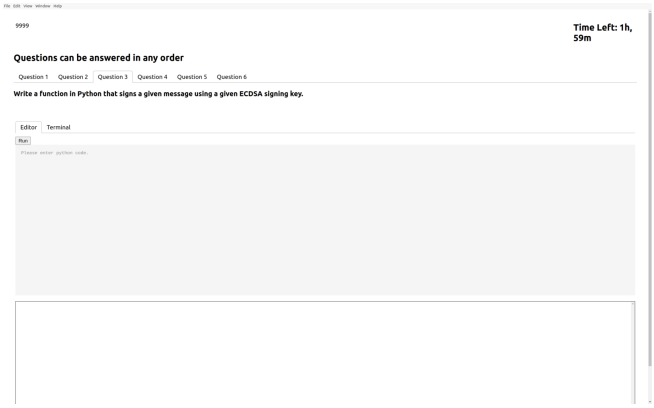
Question	Variable	Treatment	Reference	coef	std err	z	P> |z|	B-H crit
Q1	Group	Experiment	Control	-2.1437	0.906	-2.367	0.018	0.01
	Security Class	No	Yes	-1.4325	0.800	-1.790	0.073	0.02
	Student	No	Yes	0.7689	1.093	0.704	0.482	0.03
	Years Programming			-1.5640	2.080	-0.752	0.452	0.04
Q2	Group	Experiment	Control	-2.0244	1.460	-1.386	0.166	0.02
	Security Class	No	Yes	-0.2831	1.315	-0.215	0.830	0.04
	Student	No	Yes	-41.6569	3.99e+07	-1.04e-06	1.000	0.05
	Years Programming			12.8389	7.914	1.622	0.105	0.03
Q3	Group	Experiment	Control	-0.5404	0.932	-0.580	0.562	0.05
	Security Class	No	Yes	-1.9371	0.882	-2.197	0.028	0.01
	Student	No	Yes	-9.6136	4.884	-1.968	0.049	0.01
	Years Programming			12.3537	5.429	2.275	0.023	0.01
Q4	Group	Experiment	Control	-0.8841	0.816	-1.084	0.279	0.04
	Security Class	No	Yes	-0.0428	0.756	-0.057	0.955	0.05
	Student	No	Yes	0.0527	0.985	0.054	0.957	0.04
	Years Programming			0.7150	1.923	0.372	0.710	0.05
Q5	Group	Experiment	Control	0.9709	0.852	1.140	0.254	0.03
	Security Class	No	Yes	1.3595	0.938	1.449	0.147	0.03
	Student	No	Yes	-9.4088	5.105	-1.843	0.065	0.02
	Years Programming			11.3443	5.783	1.962	0.050	0.02
All	Group	Experiment	Control	-0.6315	0.331	-1.908	0.056	
	Security Class	No	Yes	-0.6453	0.328	-1.966	0.049	
	Student	No	Yes	-0.8168	0.515	-1.585	0.113	
	Years Programming			1.7321	0.917	1.890	0.059	

Table 6. Logistic Regression Table. The B-H crit column contains the critical values needed for statistical significance after the Benjamini-Hochberg correction.

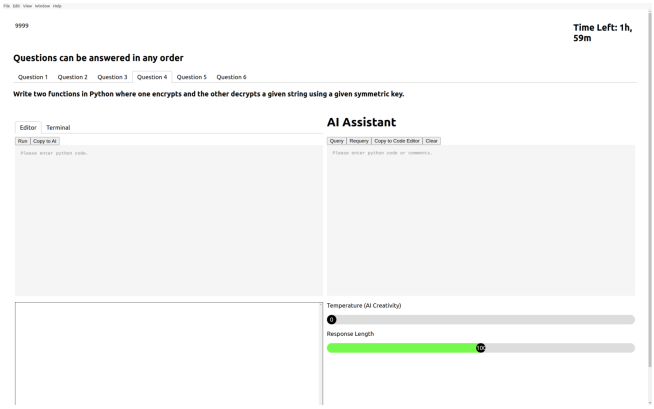
- What year did you last take a programming class in?
- For how many years have you been programming?
- How did you primarily learn how to program? (In a university / in an online learning program / in a professional certification program / on the job)
- How often do you pair program? (Frequently / occasionally / never)
- Have you ever taken a computer security class? (Yes/No)
- At what level did you take your computer security class? (Undergraduate level/graduate level/online learning/professional training)
- When did you last take a computer security class?
- Do you have experience working in computer security or privacy outside of school? (Professional / hobby / none)
- Which range below includes your age? (Under 18, 18-25, every 10 years until 85, 85 or older)
- How do you describe your gender identity? (Male/Trans Male/Female/Trans Female/Gender Non-conforming/Free response)
- What country did you (primarily) grow up in?
- What is your native language (mother tongue)?

8.8. UI Figures

Figure 5 contains screenshots of the User Interface for the experiment and control groups while a question is being solved.



(a) Control Group



(b) Experiment Group

Figure 5. Screenshots of the UI when solving one of the six questions for both participant groups.

8.9. AI vs non-AI Experiment

Table 7 shows where mistakes were attributed to within the experiment group. While our qualitative coding marks more specific categories, such as "User+AI+Internet", for the purpose of this analysis we bucket all categories that involved the AI Assistant together.

	Mistake	AI	non-AI
Q1	auth	58%	9%
	padding	12%	0%
	trivial	36%	6%
	mode	9%	0%
	library	0%	0%
Q2	random	48%	15%
Q3	parent	61%	15%
	symlink	73%	15%
Q4	sql injection	30%	6%
Q5	buffer overflow	12%	6%
	local pointer	9%	9%
	int overflow	15%	3%

Table 7. Percentage of mistakes made within the experiment group, broken down by the originator of the mistake (AI vs non-AI).

8.10. Demographics

Table 8 and Table 9 contain more detailed demographics on the participant population for the experiment and control groups respectively.

8.11. User Interaction Cycle Example

8.12. Regression Tables

Table 6 contains the data for the logistic regression used in Section 4. Data was bucketed as follows. For Q1, "Secure" and "Partially Secure" answers were grouped as secure. "Insecure" answers were grouped as insecure. For Q2, "Secure"

	education	student	type	experience	years	security	age	gender	country	language
23	A	Yes	U	Professional	3	No	18 - 24	Trans Female	US	English
106	B	Yes	G	Professional	5	No	18 - 24	Male	China	Chinese
1001	HS	Yes	U	Professional	7	Yes	18 - 24	Female	US	English
1003	M	Yes	G	Professional	15	No	25 - 34	No Answer	US	English
1004	M	Yes	G	Hobby	12	No	18 - 24	Male	Portugal	Portuguese
1008	M	No			44	No	65 - 74	Male	India	Telugu
1010	D	No			48	Yes	55 - 64	Male	US	English
1014	HS	Yes	U	Hobby	2	No	18 - 24	Female	China	Chinese
1015	HS	Yes	U	Professional	5	No	18 - 24	Male	US	English
1016	B	No			4	No	18 - 24	Male	US	English
1017	B	No			4	Yes	18 - 24	Male	US	English
1020	HS	Yes	U	Hobby	3	No	18 - 24	Female	US	Mongolian
1022	HS	Yes	U	Professional	3	No	18 - 24	Male	US	English
1023	HS	Yes	U	Hobby	4	No	18 - 24	Male	Malaysia	English
1024	B	Yes	G	Professional	3	Yes	25 - 34	Male	Indonesia	Kannada
1027	HS	Yes	U	None	3	No	18 - 24	Male	US	English
1028	HS	Yes	U	Professional	4	No	18 - 24	Female	China	Chinese
1029	HS	Yes	U	Hobby	3	No	18 - 24	Male	Myanmar	Burmese
1031	HS	Yes	U	Professional	4	No	18 - 24	Male	US	English
1032	HS	Yes	U	Professional	4	No	18 - 24	Male	US	Chinese
1033	HS	Yes	U	Hobby	10	No	18 - 24	Male	US	English
1034	HS	Yes	U	Hobby	2	Yes	18 - 24	Male	US	English
1036	A	Yes	U	Hobby	3	No	18 - 24	Female	India	Hindi
1037	B	No			7	Yes	18 - 24	Female	US	English
1038	HS	Yes	U	None	5	No	18 - 24	Female	India	Kannada
1040	M	No			7	No	18 - 24	Male	India	
1041	B	Yes	U	Professional	8	Yes	18 - 24	Male	US	English
1042	HS	Yes	U		2	No	18 - 24	Female	US	Tamil
1043	HS	Yes	U	Hobby	1	No	18 - 24	Male	China	Chinese
1045	HS	Yes	U	None	1	No	18 - 24	Female	India	Hindi
1046	HS	Yes	U	Professional	3	Yes	18 - 24	Female	India	Hindi
2001	B	Yes	G	Professional	9	Yes	18 - 24	Male	US	Chinese
2003	D	Yes	G	Professional	15	Yes	25 - 34	Male	US	English

Table 8. Experiment Participants. The column education contains the highest level of education that a participant has achieved, where A is an Associates degree, B is a Bachelors degree, HS, is a high school deploma, and D is a Doctoroal or Professional Agree. The column type contains the type of student, where U is undergrad and G is graduate. The column years contains the number of years of programming experience that a participant has. The column security contains if the participant has taken a security class.

	education	student	type	experience	years	security	age	gender	country	language
22	HS	Yes	U	None	5	No	18 - 24	Male	US	English
177	B	Yes	G	Hobby	3	Yes	18 - 24	Female		
178	HS	Yes	U	Professional	7	No	18 - 24	Male	Brazil	Portuguese
1002	M	Yes	G	Professional	13	Yes	25 - 34	Male	China	Chinese
1005	HS	Yes	U	Professional	10	Yes	18 - 24	Male	US	English
1009	HS	Yes	U	Hobby	8	Yes	18 - 24	Trans Male	US	English
1012	HS	Yes	U	Hobby	1	No	18 - 24	Female	China	Chinese
1013	HS	Yes	U	Hobby	3	No	18 - 24	Male	Hong Kong	Chinese
1018	B	Yes	U	Professional	3	No	18 - 24	Female	China	Chinese
1019	HS	Yes	U	Hobby	13	No	18 - 24	Male	US	English
1030	HS	Yes	U	Professional	5	No	18 - 24	Male	US	English
1035	B	No			8	No	18 - 24	Male	US	English
1039	HS	Yes	U	Professional	4	No	18 - 24	Male	US	English
2002	B	Yes	G	Professional	7	No	18 - 24	Male	US	English

Table 9. Control Participants. The column education contains the highest level of education that a participant has achieved, where A is an Associates degree, B is a Bachelors degree, HS, is a high school deploma, and D is a Doctoroal or Professional Agree. The column type contains the type of student, where U is undergrad and G is graduate. The column years contains the number of years of programming experience that a participant has. The column security contains if the participant has taken a security class.

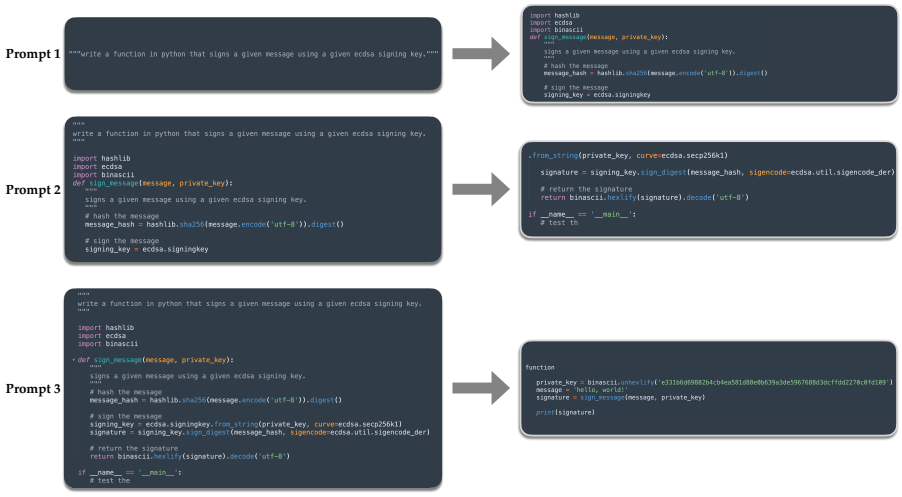


Figure 6. An example interaction with the AI assistant where the user, Participant 1036, enters a cycle and repeatedly uses the model's output (right) as the text for their next prompt, trusting that `ecdsa` is an appropriate library to use.

answers were grouped as secure. “Partially Secure” and “Insecure” answers were grouped as insecure. For Q3, “Secure” and “Partially Secure” answers were grouped as secure. “Insecure” answers were grouped as insecure. For Q4, “Secure” answers were grouped as secure and “Insecure” answers were grouped as insecure. For Q5, “Secure”, “RC”, and “DoS” answers were grouped as secure. “Partially Secure” and “Insecure” answers were grouped as insecure. “Partially Secure” answers were placed into different buckets for different questions due to their varying severity. Note that while this table reports results for the effect of the experiment/control groups, we determine statistical significance of this treatment for particular security buckets (e.g. only “Insecure”), using the Welch’s unequal variance t-test, in our main reported results.