NYU TANDON SCHOOL OF ENGINEERING

Lost at C

Security Implications of Large Language Model Code Assistants

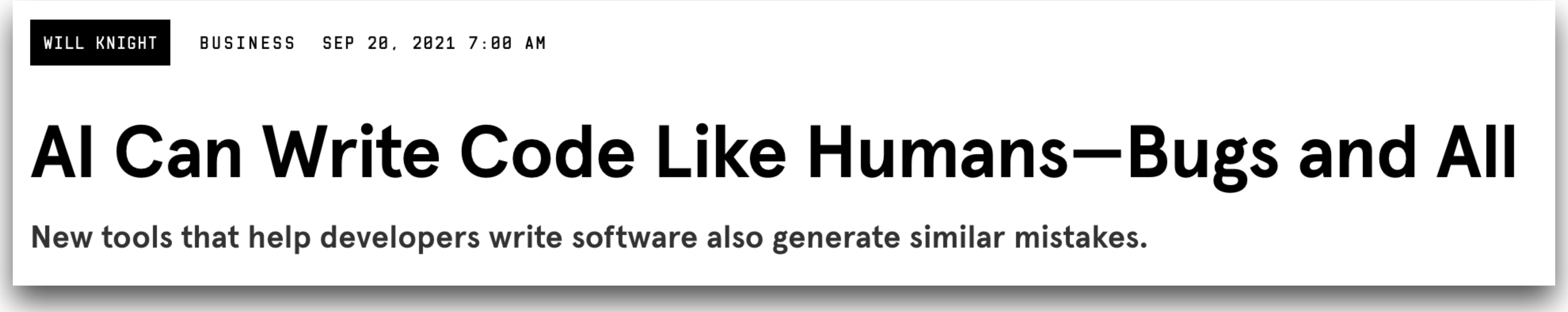
Brendan Dolan-Gavitt

In collaboration with: Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, and Siddharth Garg

CENTER FOR CYBER SECURITY

How Secure is the Code LLMs Write?

How Secure is the Code LLMs Write?

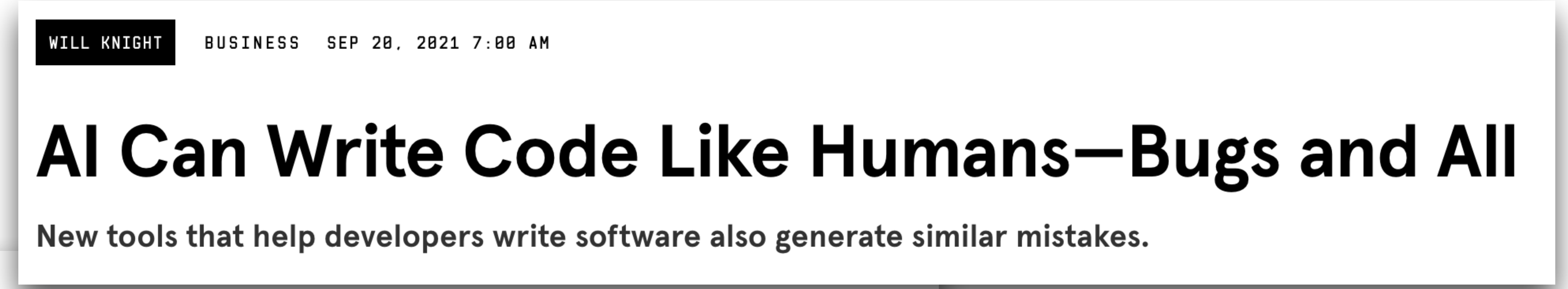


WILL KNIGHT BUSINESS SEP 20, 2021 7:00 AM

AI Can Write Code Like Humans—Bugs and All

New tools that help developers write software also generate similar mistakes.

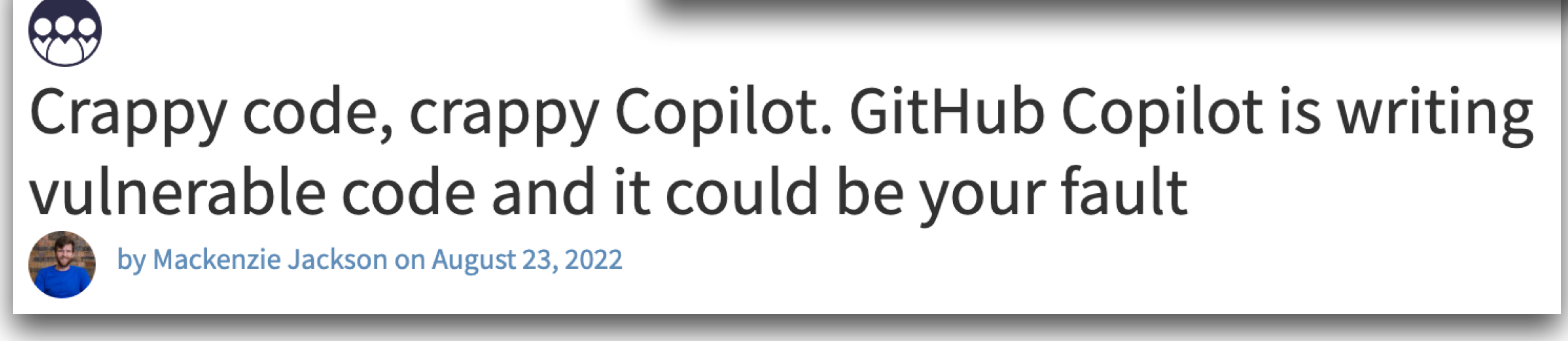
How Secure is the Code LLMs Write?



WILL KNIGHT BUSINESS SEP 20, 2021 7:00 AM

AI Can Write Code Like Humans—Bugs and All

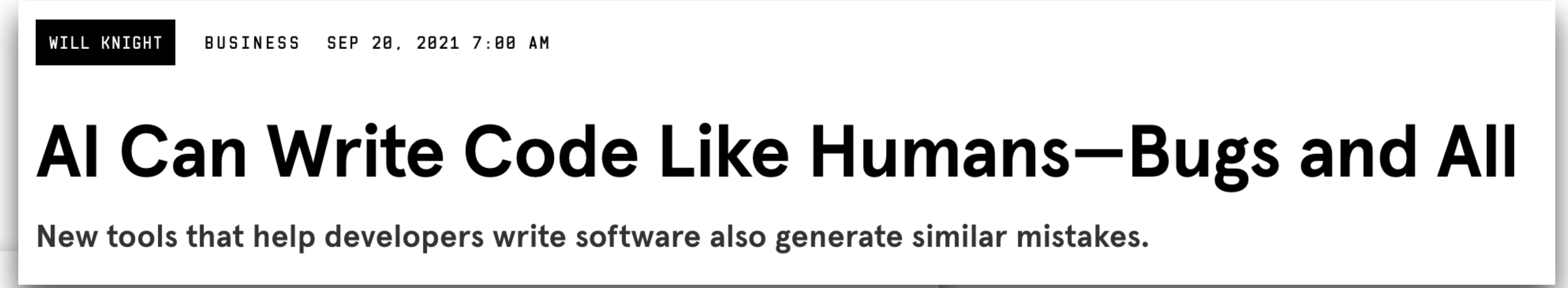
New tools that help developers write software also generate similar mistakes.



Crappy code, crappy Copilot. GitHub Copilot is writing vulnerable code and it could be your fault

by Mackenzie Jackson on August 23, 2022

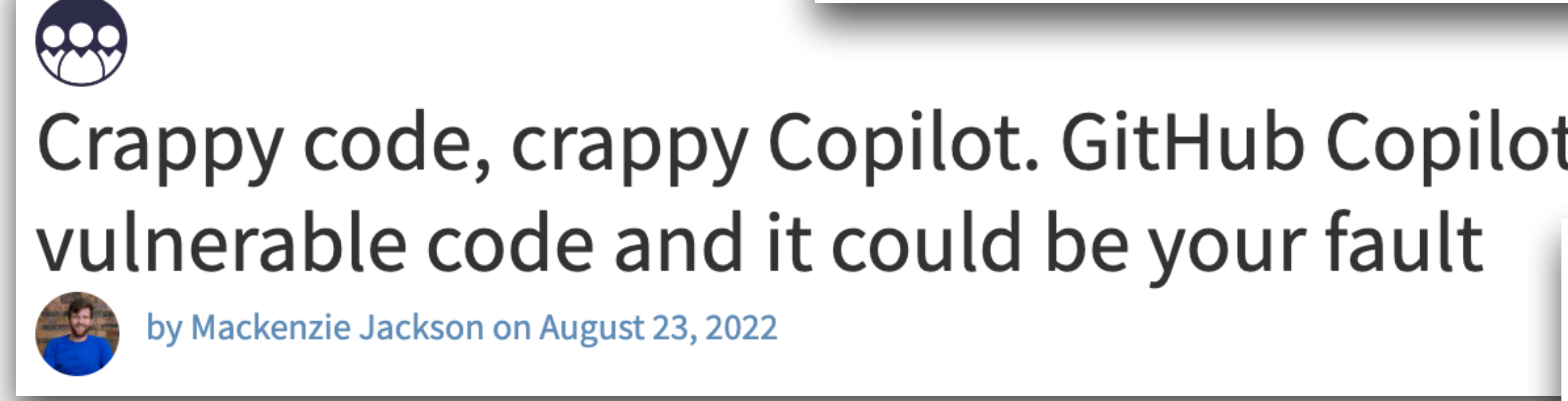
How Secure is the Code LLMs Write?



WILL KNIGHT BUSINESS SEP 20, 2021 7:00 AM

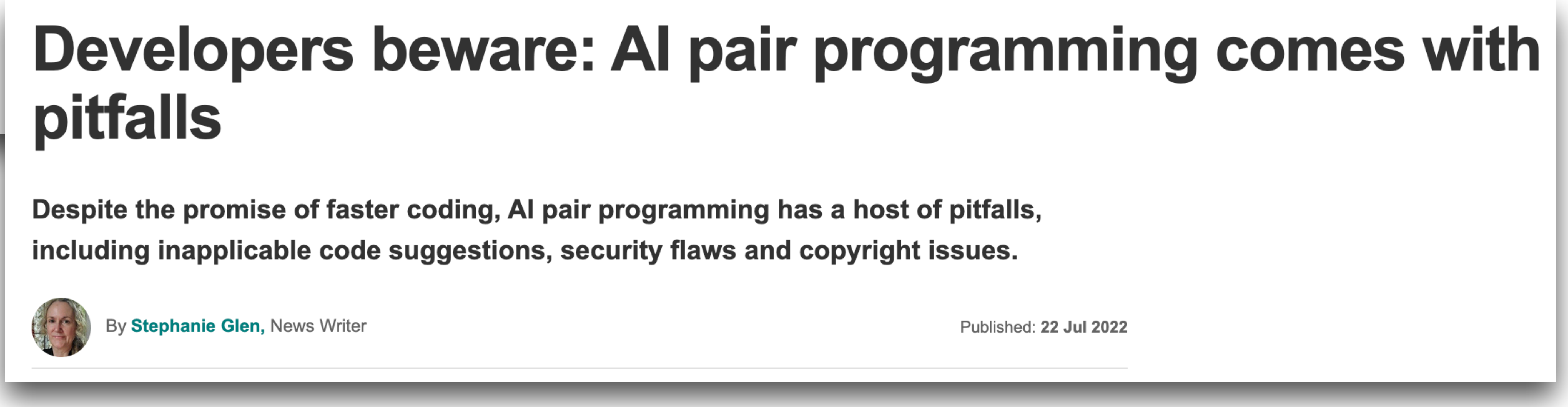
AI Can Write Code Like Humans—Bugs and All

New tools that help developers write software also generate similar mistakes.



Crappy code, crappy Copilot. GitHub Copilot is writing vulnerable code and it could be your fault

by Mackenzie Jackson on August 23, 2022



Developers beware: AI pair programming comes with pitfalls

Despite the promise of faster coding, AI pair programming has a host of pitfalls, including inapplicable code suggestions, security flaws and copyright issues.

By Stephanie Glen, News Writer

Published: 22 Jul 2022

How Secure is the Code LLMs Write?

WILL KNIGHT BUSINESS SEP 20, 2021 7:00 AM

AI Can Write Code Like Humans—Bugs and All

New tools that help developers write software also generate similar mistakes.

Crappy code, crappy Copilot. GitHub Copilot is writing vulnerable code and it could be your fault

by Mackenzie Jackson on August 23, 2022

Developers beware: AI pair programming comes with pitfalls

Despite the promise of faster coding, AI pair programming has a host of pitfalls, including inapplicable code suggestions, security flaws and copyright issues.

By Stephanie Glen, News Writer

Published: 22 Jul 2022

FEATURE

Why you can't trust AI-generated autocomplete code to be secure

Artificial intelligence-powered tools such as GitHub Pilot and Tabnine offer developers autocomplete suggestions that help them write code faster. How do they ensure this code is secure?

By Andrada Fiscutean

CSO | MAR 15, 2022 2:00 AM PDT

How Secure is the Code LLMs Write?

WILL KNIGHT BUSINESS SEP 20, 2021 7:00 AM

AI Can Write Code Like Humans—Bugs and All

New tools that help developers write software also generate similar mistakes.

Crappy code, crappy Copilot. GitHub Copilot is writing vulnerable code and it could be your fault

by Mackenzie Jackson on August 23, 2022

Developers beware: AI pair programming comes with pitfalls

Despite the promise of faster coding, AI pair programming has a host of pitfalls, including inapplicable code suggestions, security flaws and copyright issues.

By **Stephanie Glen,** News Writer

Published: **22 Jul 2022**

FEATURE

Why you can't trust AI-generated autocomplete code to be secure

Artificial intelligence-powered tools such as GitHub Pilot and Tabnine offer developers autocomplete suggestions that help them write code faster. How do they ensure this code is secure?

By **Andrada Fiscutean**

CSO | MAR 15, 2022 2:00 AM PDT

GitHub Copilot Security Study: 'Developers Should Remain Awake' in View of 40% Bad Code Rate

By **David Ramel** 08/26/2021

Asleep at the Keyboard

Prior work at IEEE Security and Privacy 2022

- We did a systematic study of Copilot’s code completions in security-sensitive scenarios, measuring vulnerability rates with GitHub CodeQL
- Key findings:
  - Across all scenarios, **42%** of the generated programs were vulnerable
  - Features of the **prompt**, including comments, affects the rate of vulnerable code
  - The strongest predictor of whether Copilot will produce a vulnerability is the **presence of an existing vulnerability** in the prompt

But Wait!

Some objections from Reviewer #2

- In the real world, Copilot works with human assistance
- Maybe humans would spot and fix these mistakes?
- For that matter, maybe *unassisted* humans would write bugs at the same rate!
- **Strong reject**

Research Questions

- **RQ1**: Does the AI code assistant help novice users write better code in terms of ***functionality***?

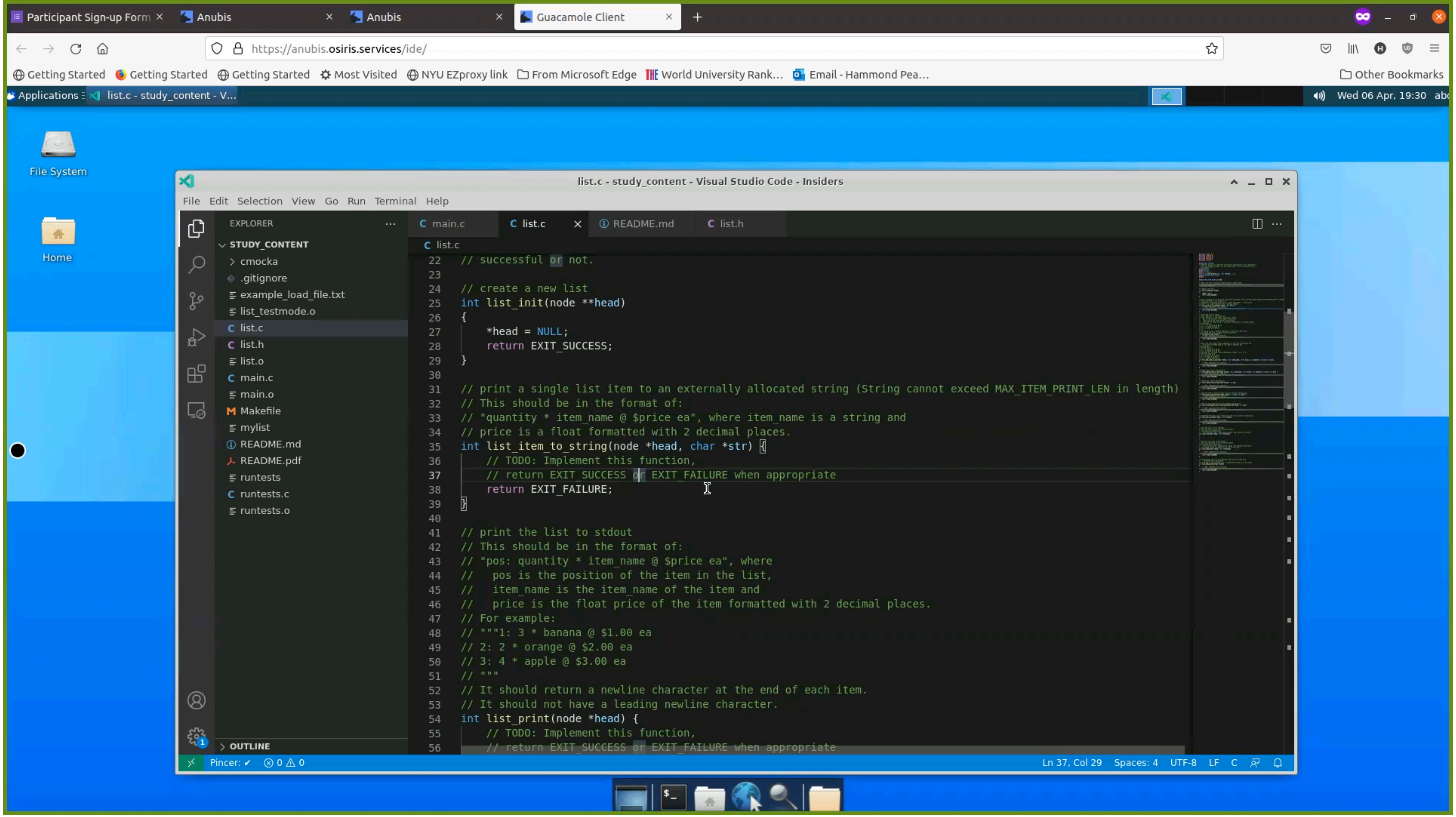
- **RQ2**: Is the code that novice users write with AI assistance more or less ***secure*** than the control group?

- **RQ3**: Are there systematic differences in the ***coding style*** of AI-assisted users and that of control group?

- **RQ4**: How do AI assisted users interact with potentially vulnerable code suggestions, i.e., where do bugs originate in an LLM-assisted system?

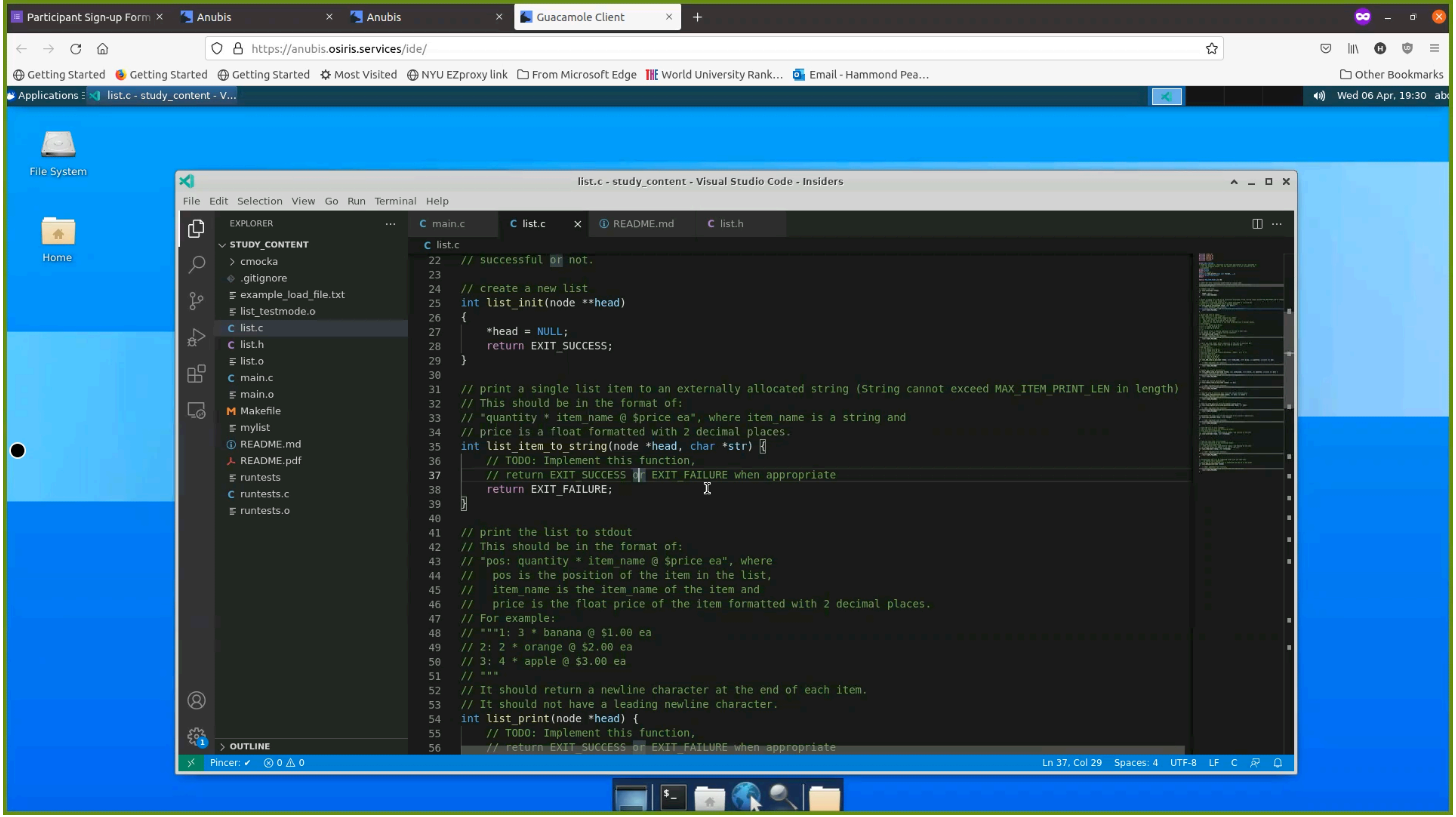
Study Environment

- **Goals**:
  - Minimize environment setup hassle
  - Log all the things
- Participants were asked to use our **Anubis** web-based IDE, which provides a VNC session to a Linux desktop with **VSCode** and a C compiler
- Created a VSCode plugin that mimics Copilot, but uses suggestions provided by the Codex API
- **Logged**: document snapshots every minute, prompt+suggestion data (including accepted/not accepted)



```c
// successful or not.

// create a new list
int list_init(node **head)
{
    *head = NULL;
    return EXIT_SUCCESS;
}

// print a single list item to an externally allocated string (String cannot exceed MAX_ITEM_PRINT_LEN in length)
// This should be in the format of:
// "quantity * item_name @ $price ea", where item_name is a string and
// price is a float formatted with 2 decimal places.
int list_item_to_string(node *head, char *str) {
    // TODO: Implement this function,
    // return EXIT_SUCCESS or EXIT_FAILURE when appropriate
    return EXIT_FAILURE;
}

// print the list to stdout
// This should be in the format of:
// "pos: quantity * item_name @ $price ea", where
//   pos is the position of the item in the list,
//   item_name is the item_name of the item and
//   price is the float price of the item formatted with 2 decimal places.
// For example:
// """1: 3 * banana @ $1.00 ea
// 2: 2 * orange @ $2.00 ea
// 3: 4 * apple @ $3.00 ea
// """
// It should return a newline character at the end of each item.
// It should not have a leading newline character.
int list_print(node *head) {
    // TODO: Implement this function,
    // return EXIT_SUCCESS or EXIT_FAILURE when appropriate
```

If you are reading these slides in PDF, you can see the video by clicking here:
https://moyix.net/~moyix/anubis.mp4



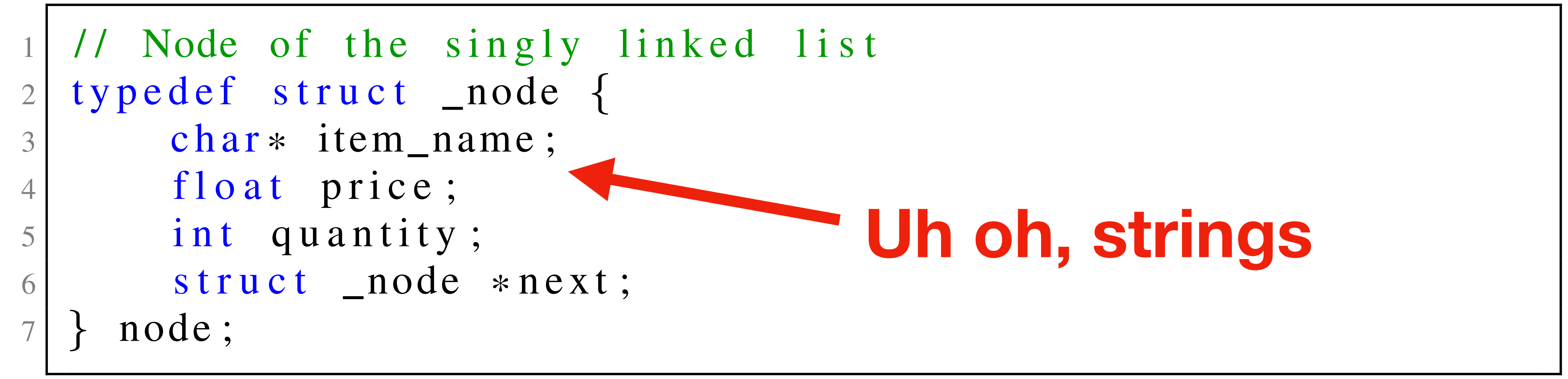
```c
// successful or not.

// create a new list
int list_init(node **head)
{
    *head = NULL;
    return EXIT_SUCCESS;
}

// print a single list item to an externally allocated string (String cannot exceed MAX_ITEM_PRINT_LEN in length)
// This should be in the format of:
// "quantity * item_name @ $price ea", where item_name is a string and
// price is a float formatted with 2 decimal places.
int list_item_to_string(node *head, char *str) {
    // TODO: Implement this function,
    // return EXIT_SUCCESS or EXIT_FAILURE when appropriate
    return EXIT_FAILURE;
}

// print the list to stdout
// This should be in the format of:
// "pos: quantity * item_name @ $price ea", where
//   pos is the position of the item in the list,
//   item_name is the item_name of the item and
//   price is the float price of the item formatted with 2 decimal places.
// For example:
// """1: 3 * banana @ $1.00 ea
// 2: 2 * orange @ $2.00 ea
// 3: 4 * apple @ $3.00 ea
// """
// It should return a newline character at the end of each item.
// It should not have a leading newline character.
int list_print(node *head) {
    // TODO: Implement this function,
    // return EXIT_SUCCESS or EXIT_FAILURE when appropriate
```

If you are reading these slides in PDF, you can see the video by clicking here:
https://moyix.net/~moyix/anubis.mp4

Study Task: “Shopping List”

The *Worst* Singly Linked List API (11 functions total)

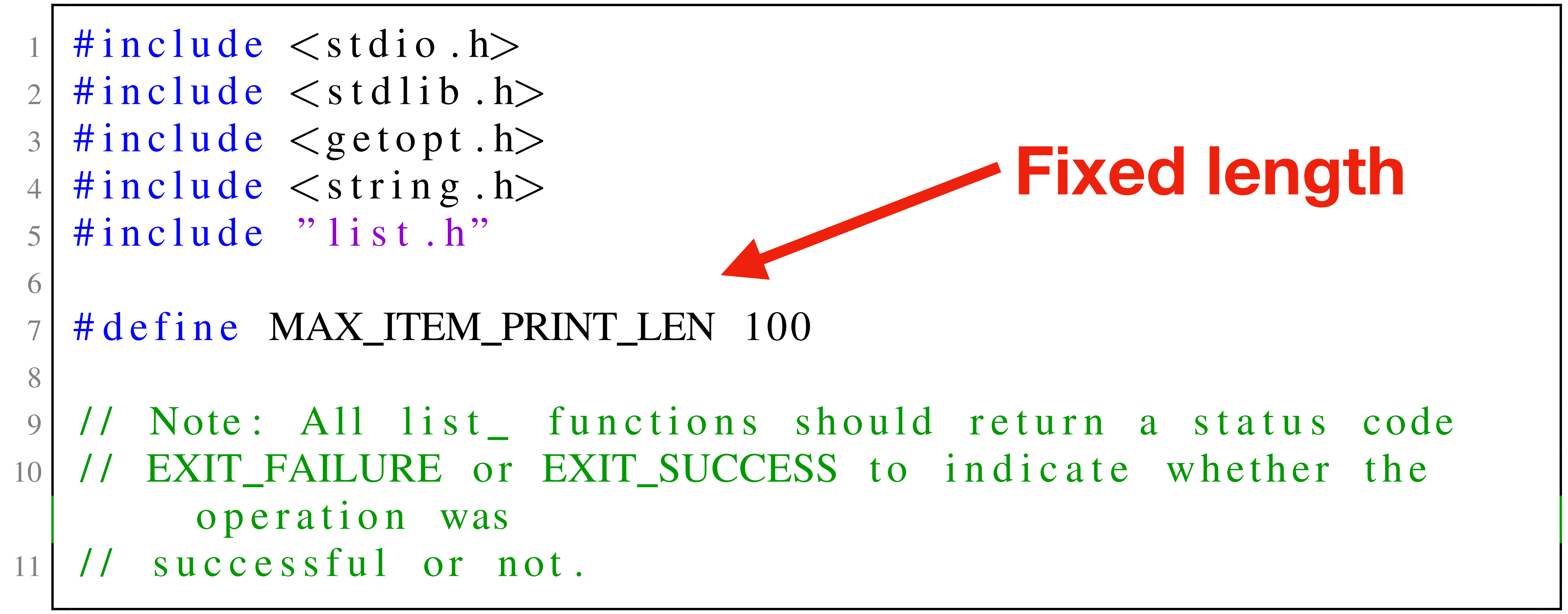
- Since we’re studying security chose C because it’s a “target-rich environment”
- We deliberately included some pitfalls in the data structure and API to further broaden the range of possible errors
- Singly linked list: lots of opportunity for pointer mistakes
- Includes a string field (buffer overflows, etc.)



```
// Node of the singly linked list
typedef struct _node {
    char* item_name;
    float price;
    int quantity;
    struct _node *next;
} node;
```

Uh oh, strings

(a) Node definition (in `list.h`)



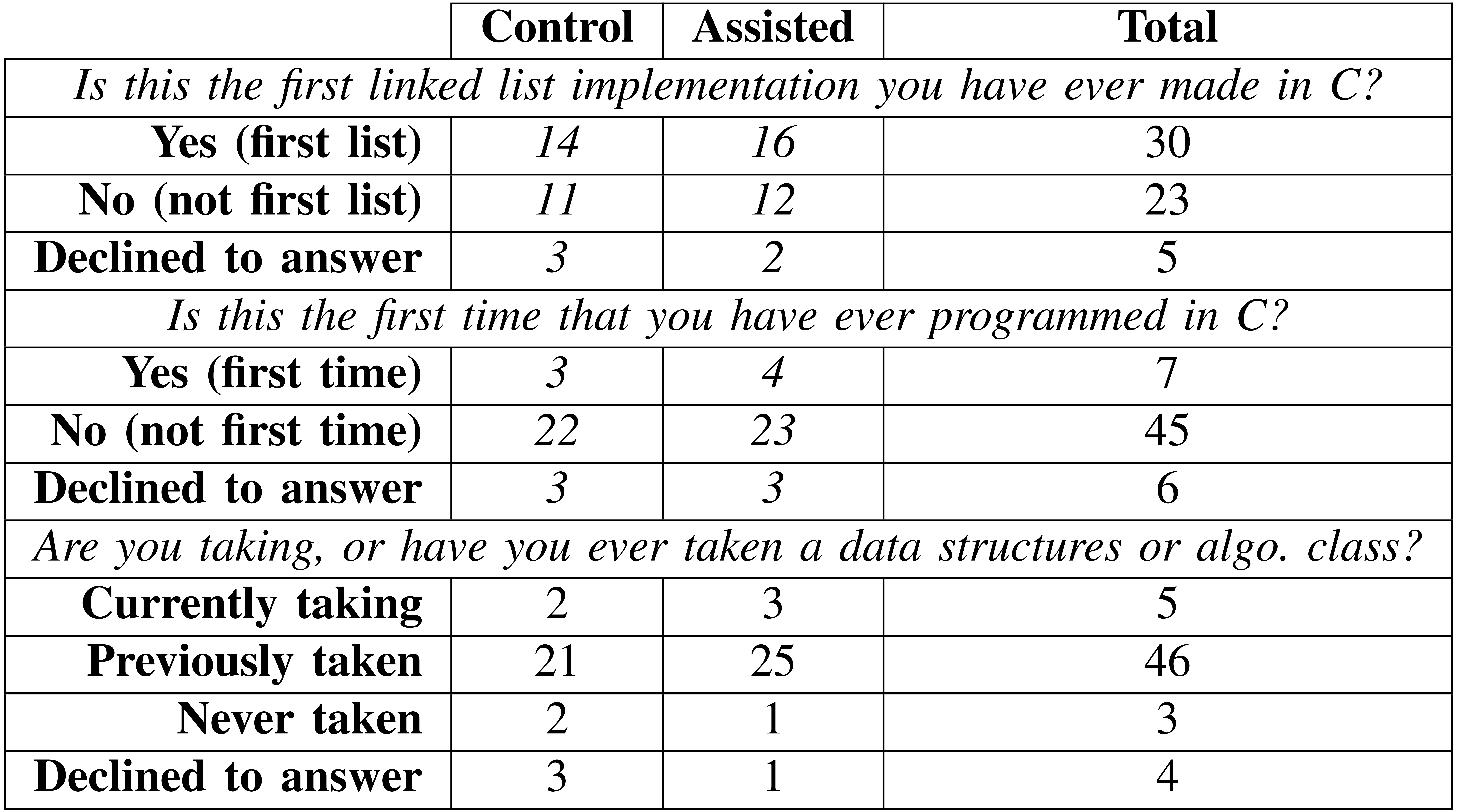
```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include ”list.h”

#define MAX_ITEM_PRINT_LEN 100

// Note: All list_ functions should return a status code
// EXIT_FAILURE or EXIT_SUCCESS to indicate whether the
     operation was
// successful or not.
```

Fixed length

(b) `#includes` and implementation hints (in `list.c`)

Participant Demographics

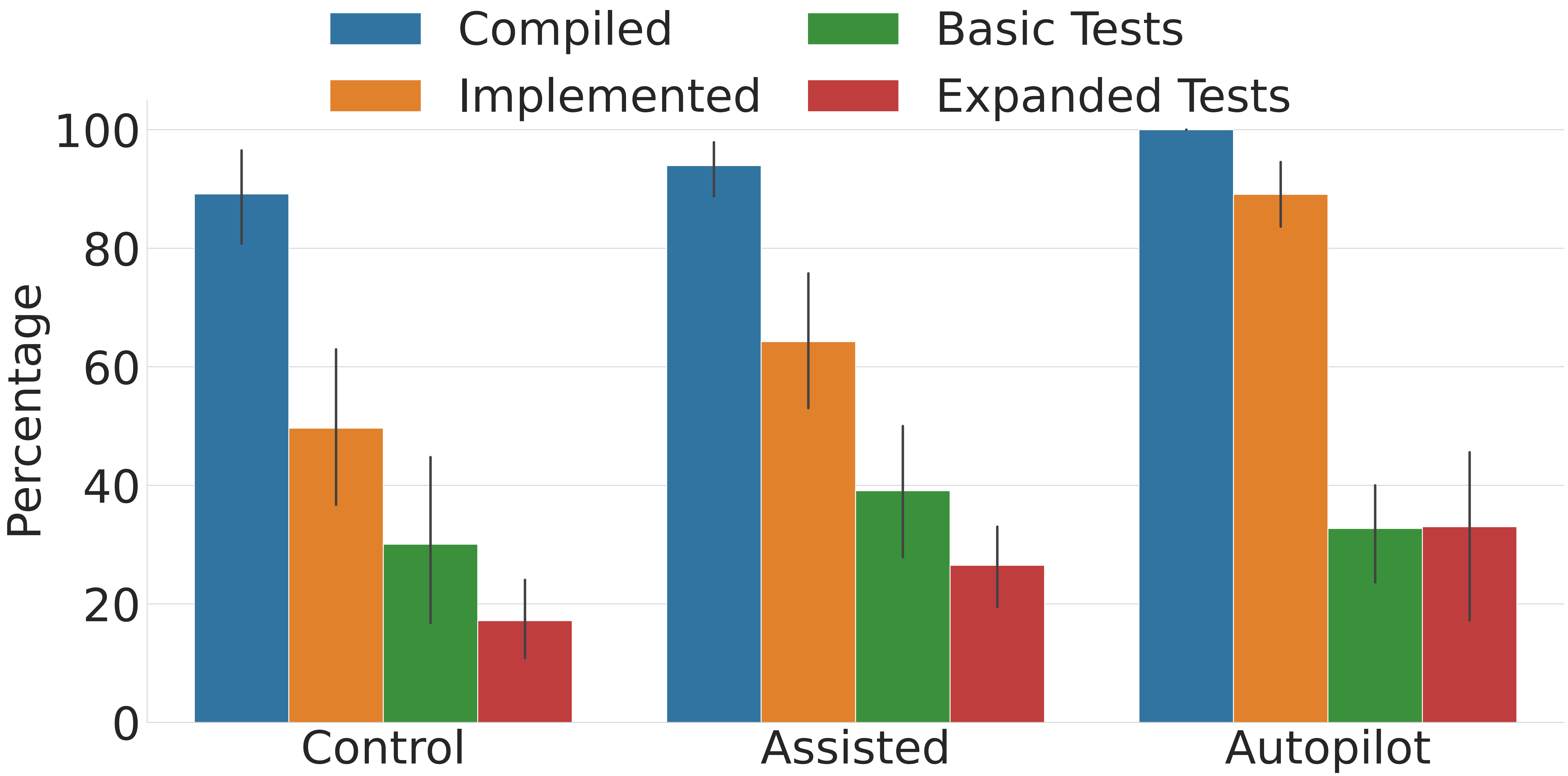
Experience Level



	Control	Assisted	Total
Is this the first linked list implementation you have ever made in C?			
Yes (first list)	*14*	*16*	30
No (not first list)	*11*	*12*	23
Declined to answer	*3*	*2*	5
Is this the first time that you have ever programmed in C?			
Yes (first time)	*3*	*4*	7
No (not first time)	*22*	*23*	45
Declined to answer	*3*	*3*	6
Are you taking, or have you ever taken a data structures or algo. class?			
Currently taking	2	3	5
Previously taken	21	25	46
Never taken	2	1	3
Declined to answer	3	1	4

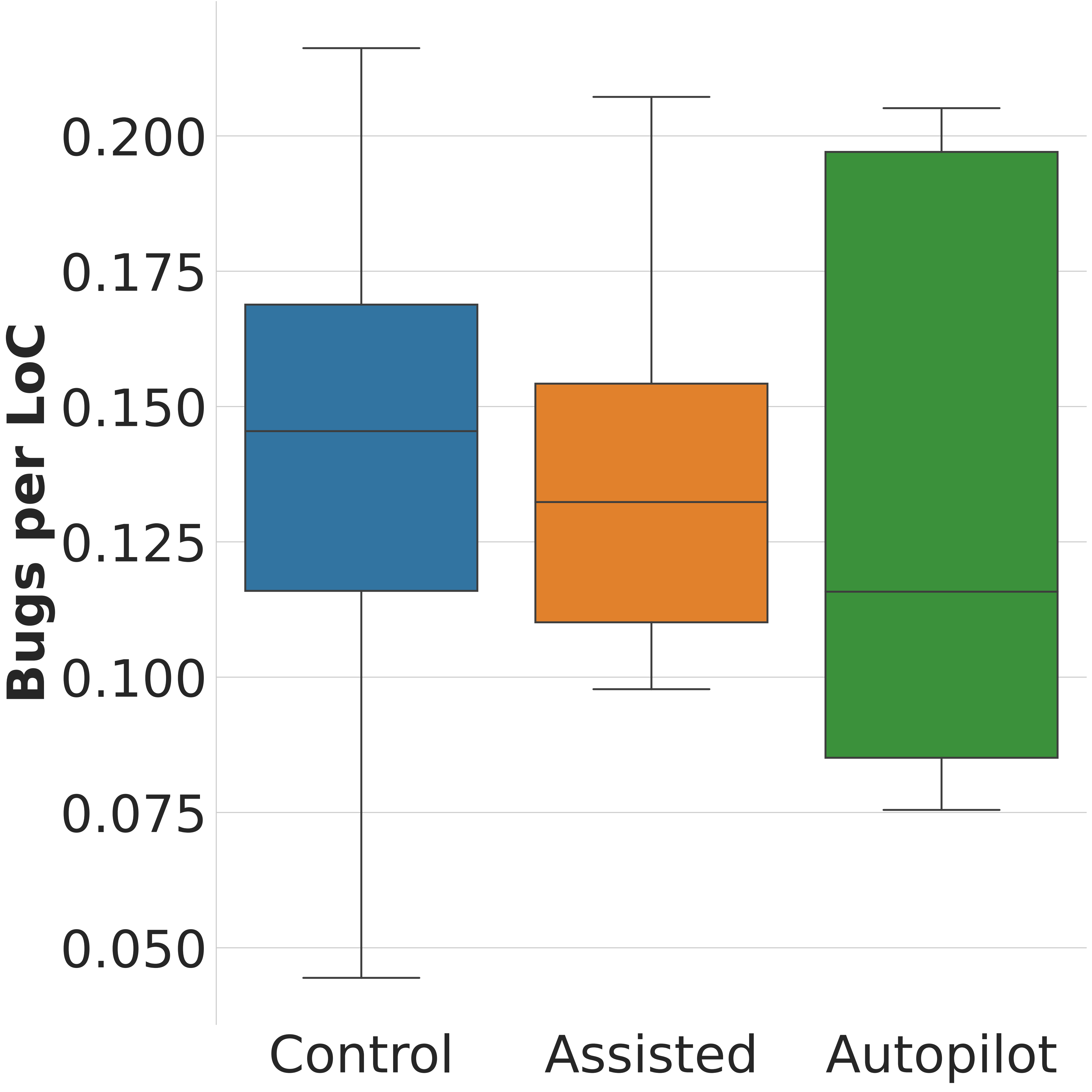
Functionality Results

Rise of the Machines

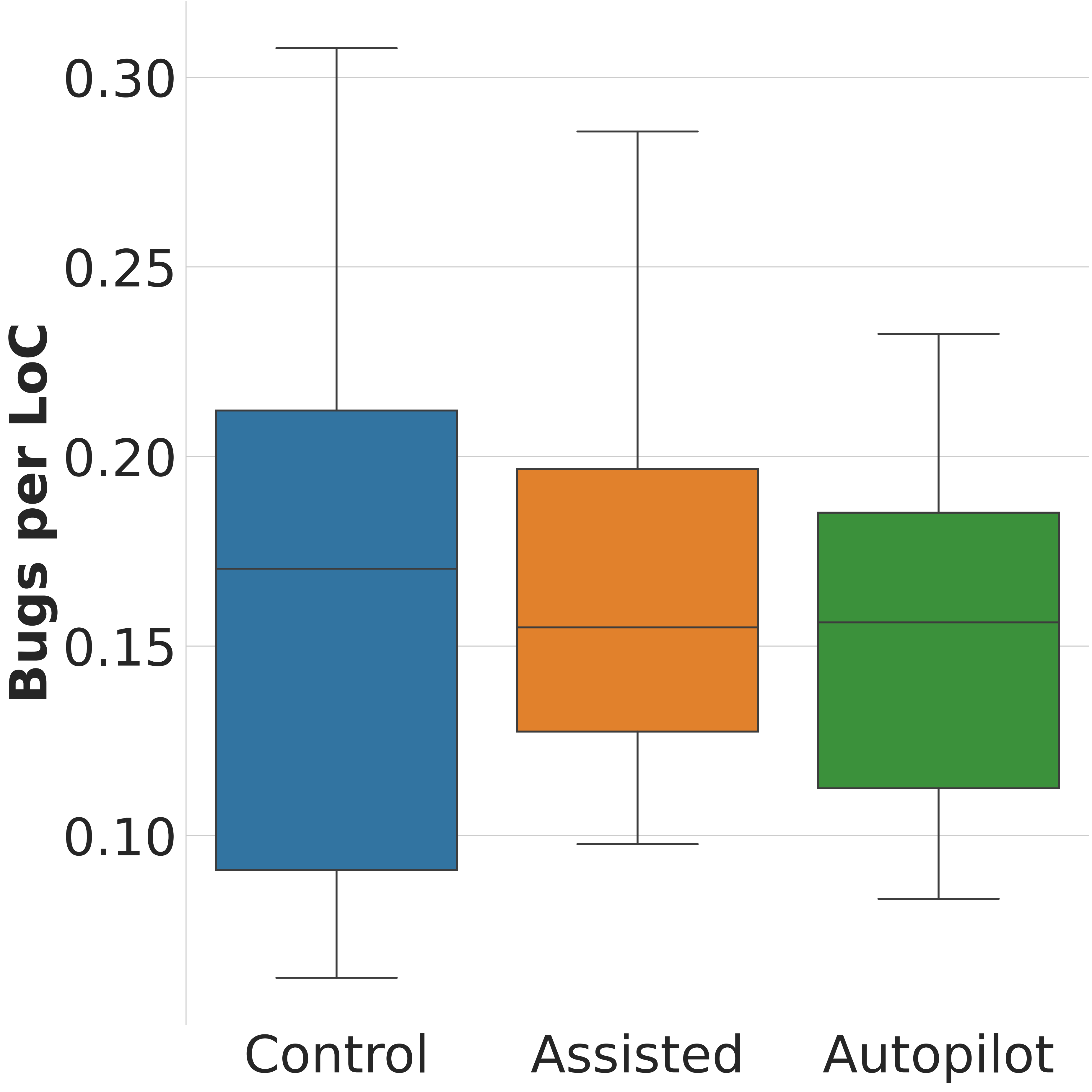


Security Results

Number of vulnerabilities per line of code



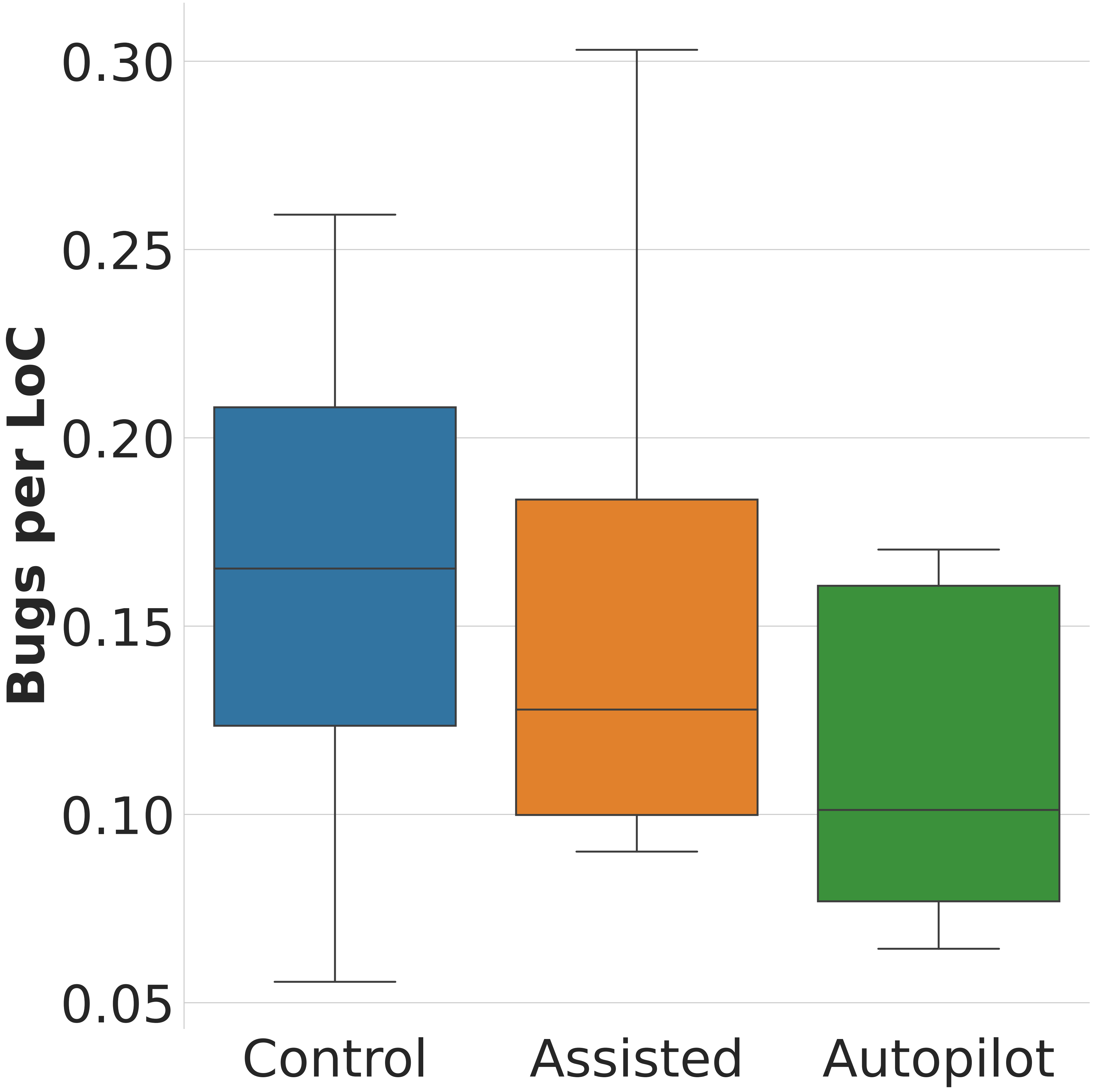
CWEs/LoC for *compiling* code



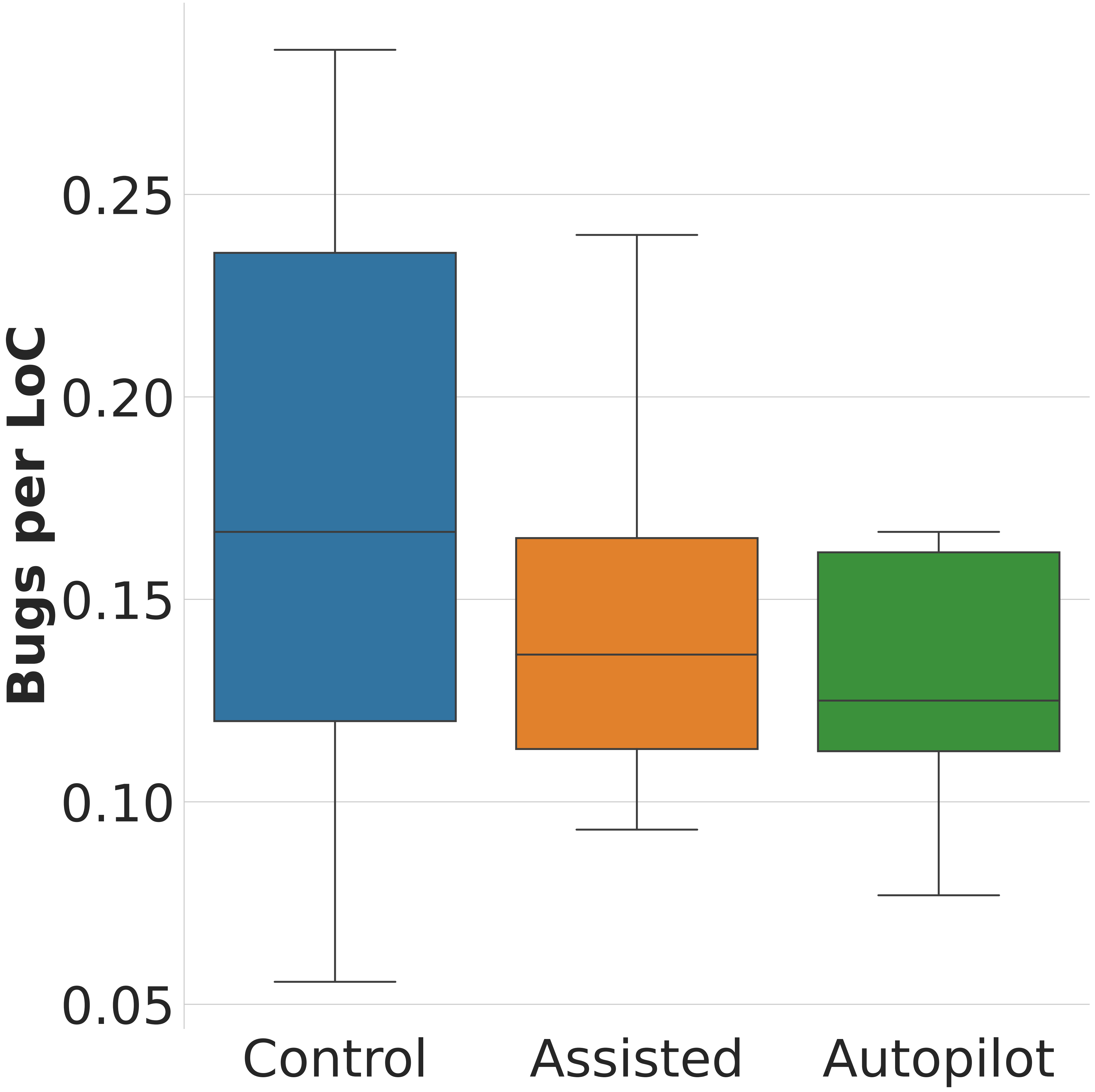
CWEs/LoC, code that *passes the basic unit test*

Security Results

Number of *severe* (MITRE Top 25) vulnerabilities per line of code

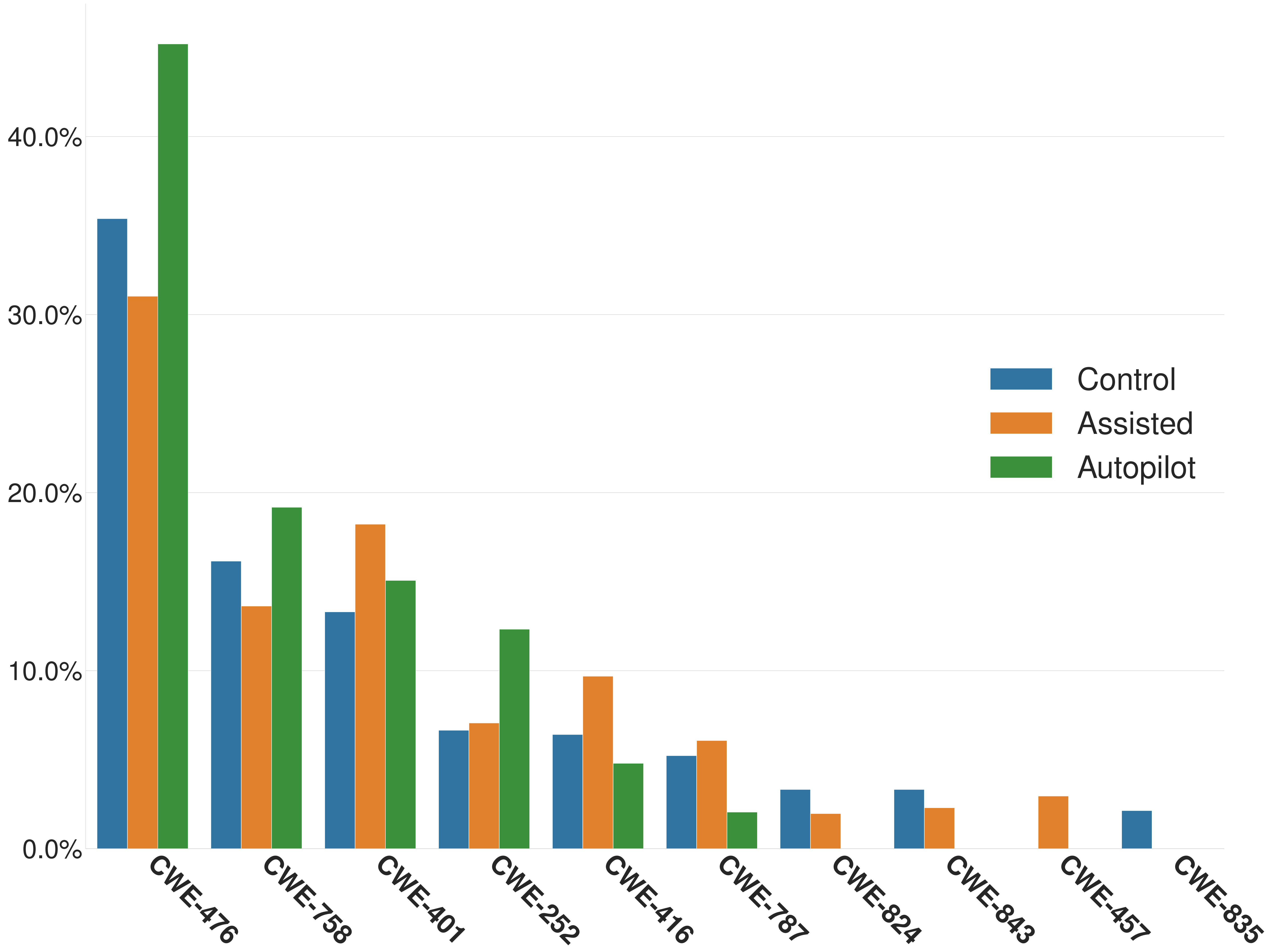


Severe CWEs/LoC for *compiling* code



Severe CWEs/LoC, code that *passes the basic unit test*

Security Results: CWEs

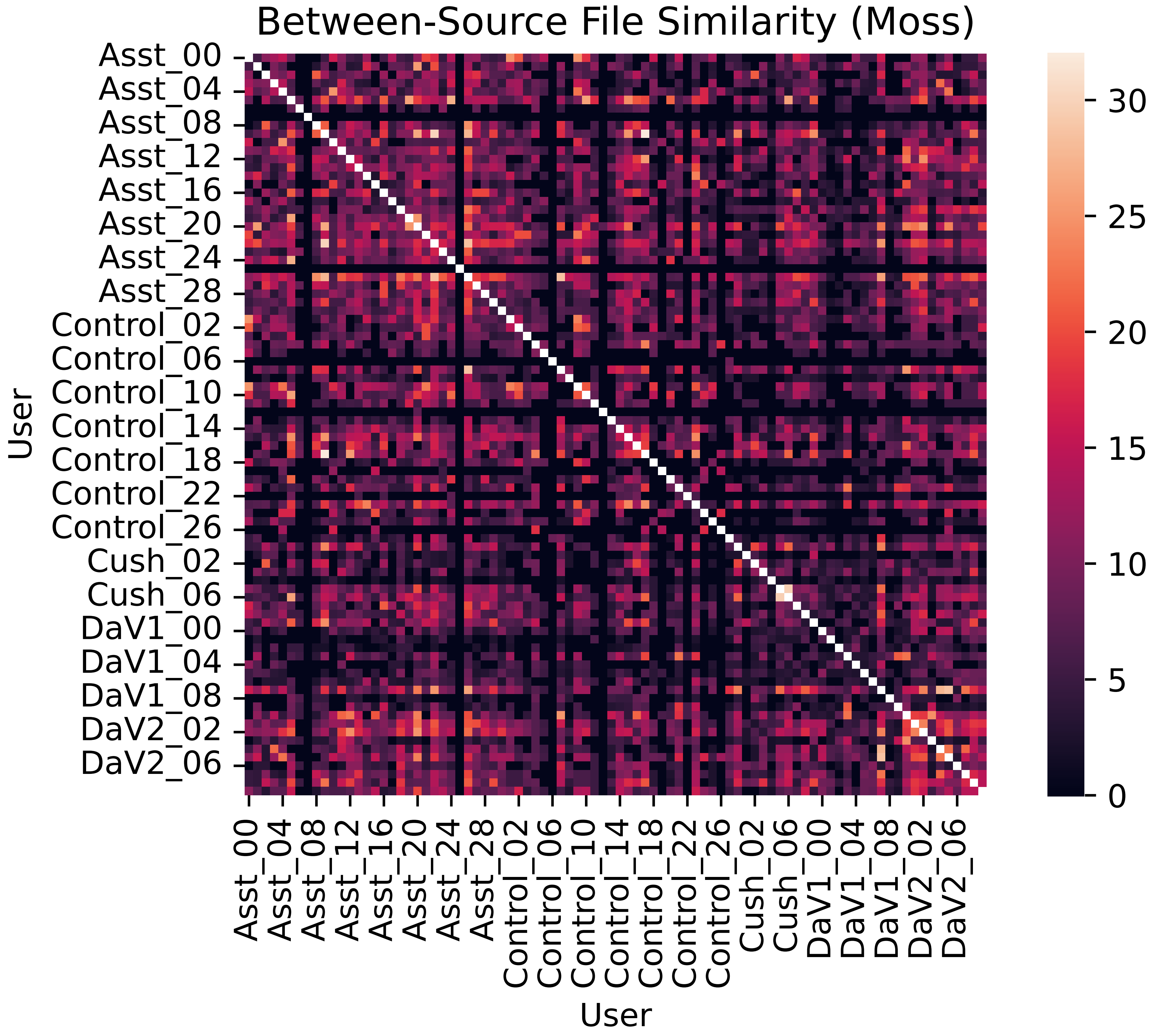


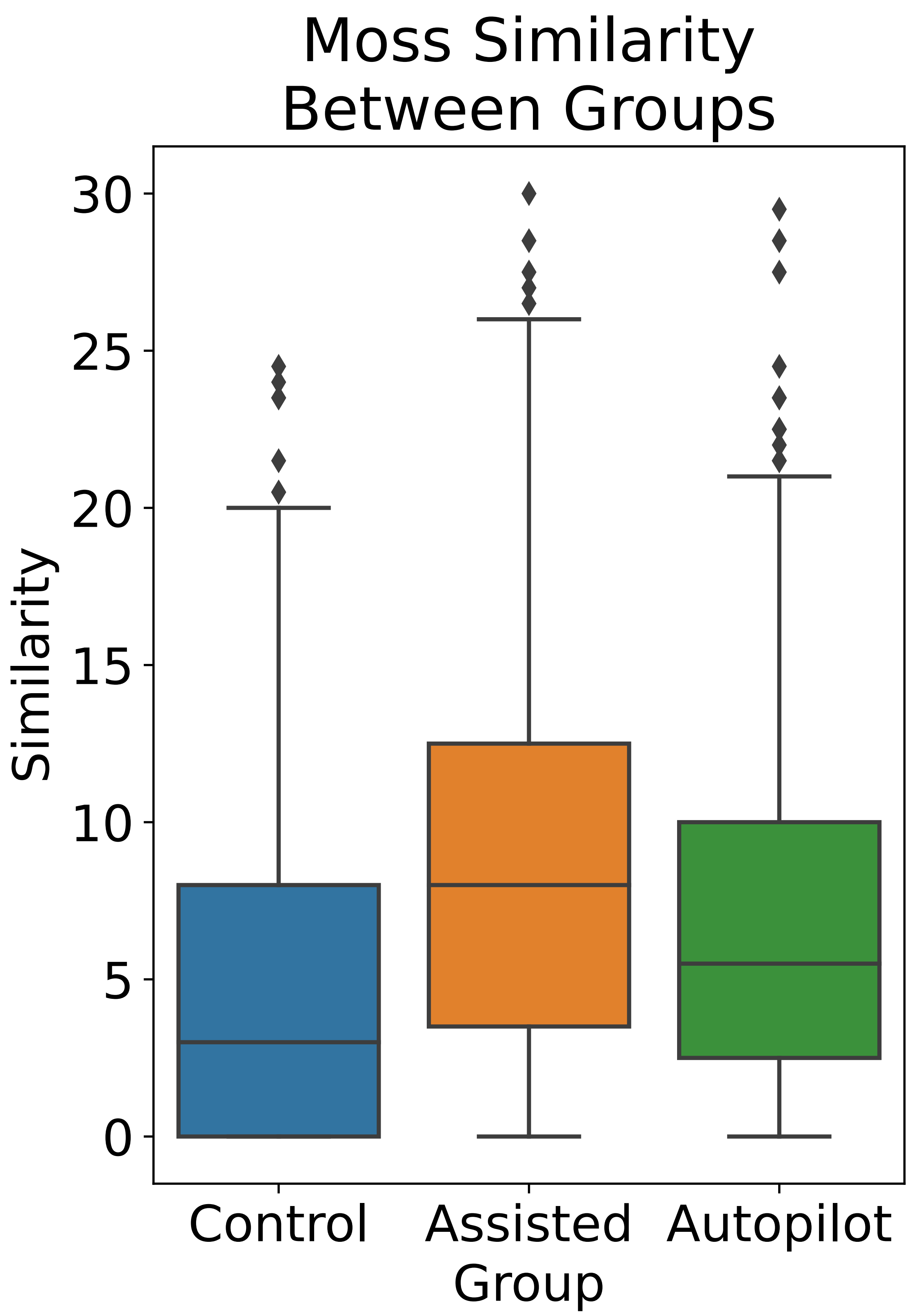
CWE-476	NULL Pointer Dereference
CWE-758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
CWE-401	Missing Release of Memory after Effective Lifetime
CWE-252	Unchecked Return Value
CWE-416	Use After Free
CWE-787	Out-of-bounds Write
CWE-457	Use of Uninitialized Variable
CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')
CWE-824	Access of Uninitialized Pointer
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')

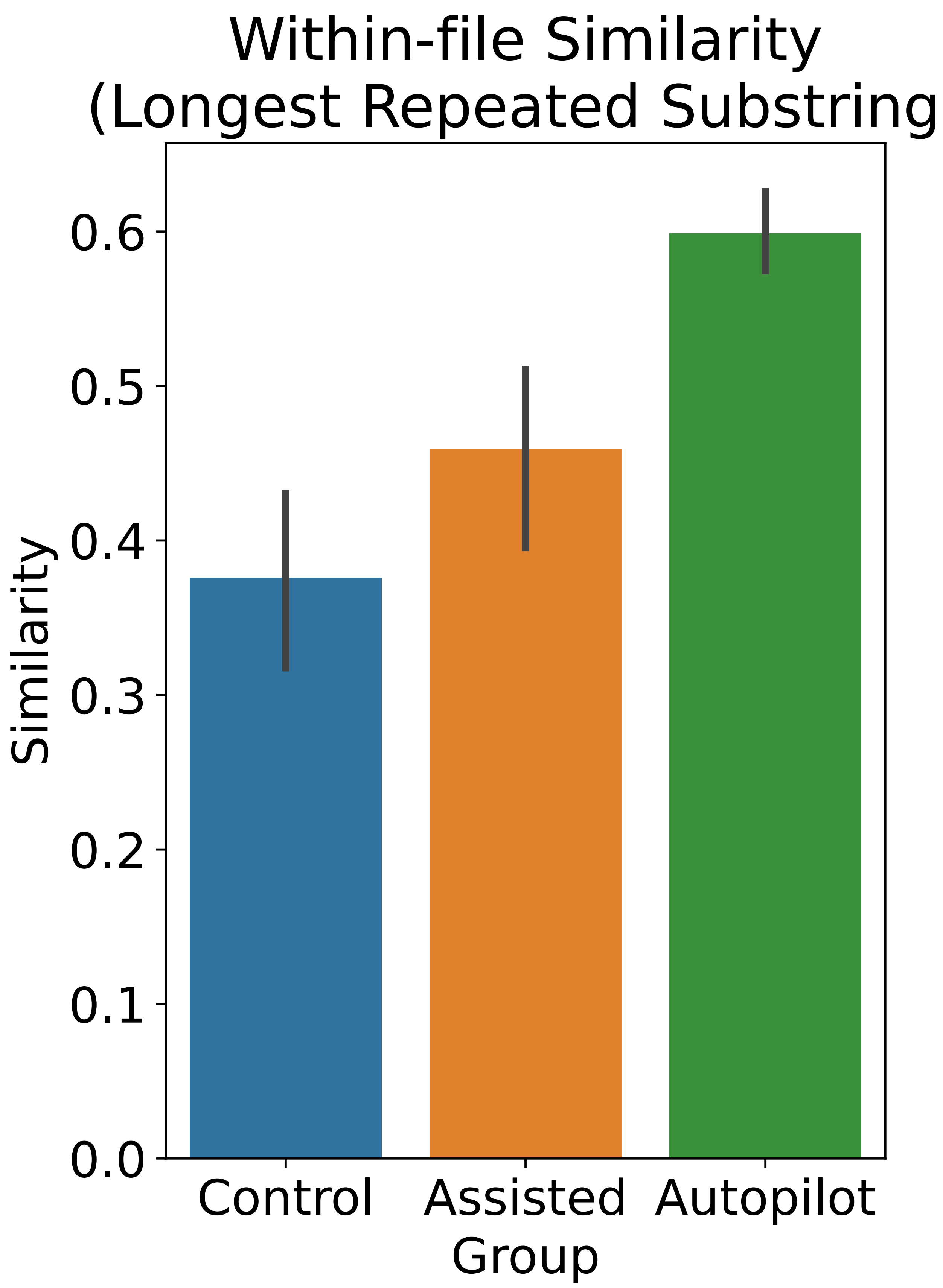
Measuring Style

- We wanted to check if there were difference in style between human and AI-assisted users
  - Can we tell if someone is using Copilot?
- We used two measures:
  - The Moss plagiarism detection tool to measure similarity between users
  - The quantity of *repeated substrings* in the file to measure similarity *within* an individual user's submission

Style Results

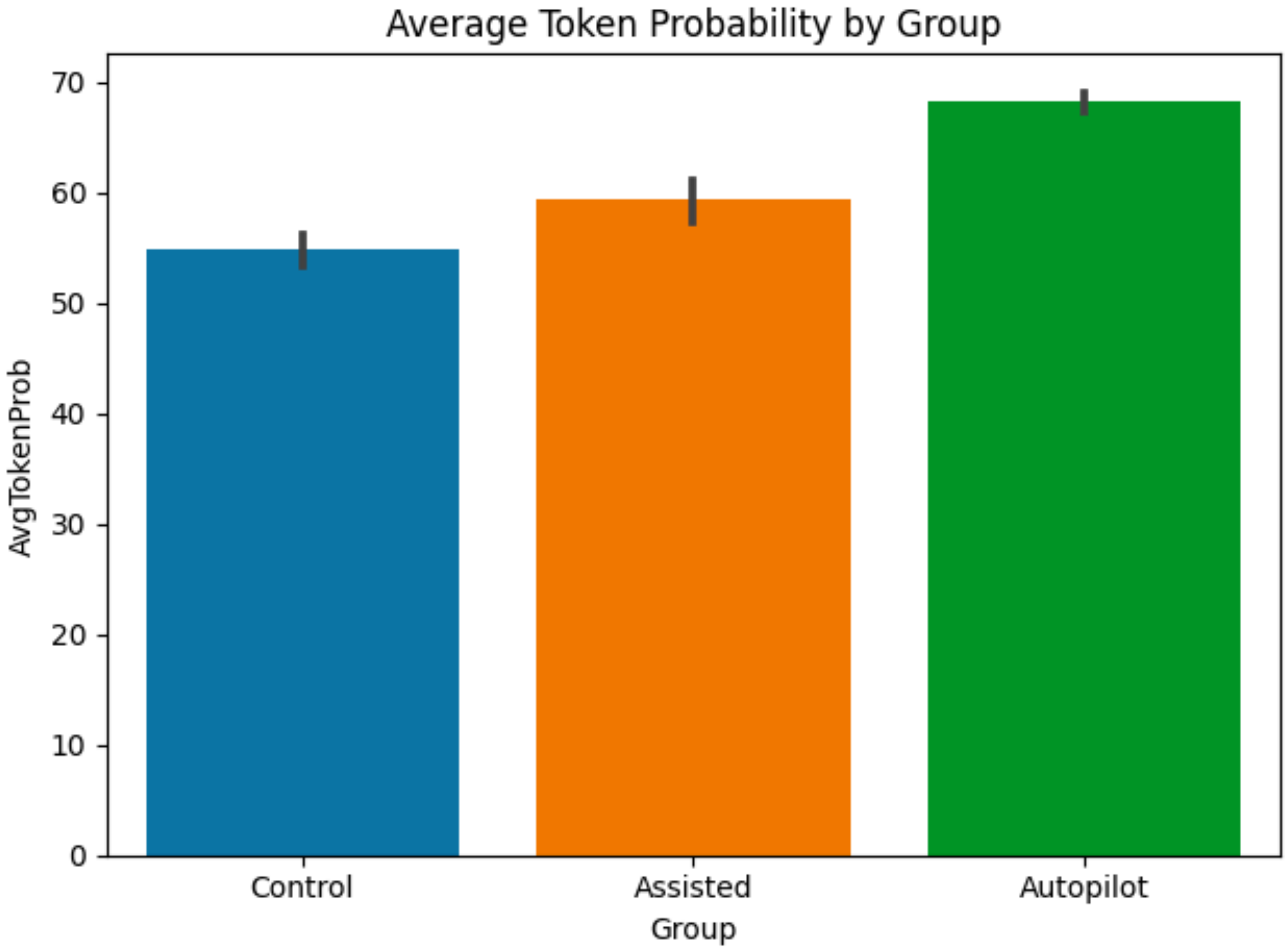

Between-Source File Similarity (Moss)


Moss Similarity
Between Groups


Within-file Similarity
(Longest Repeated Substring

Style Results (LM)

Suggested during Q&A: Use Codex to Get Prob. of Document



On the Origin of Bugs

git blame codex

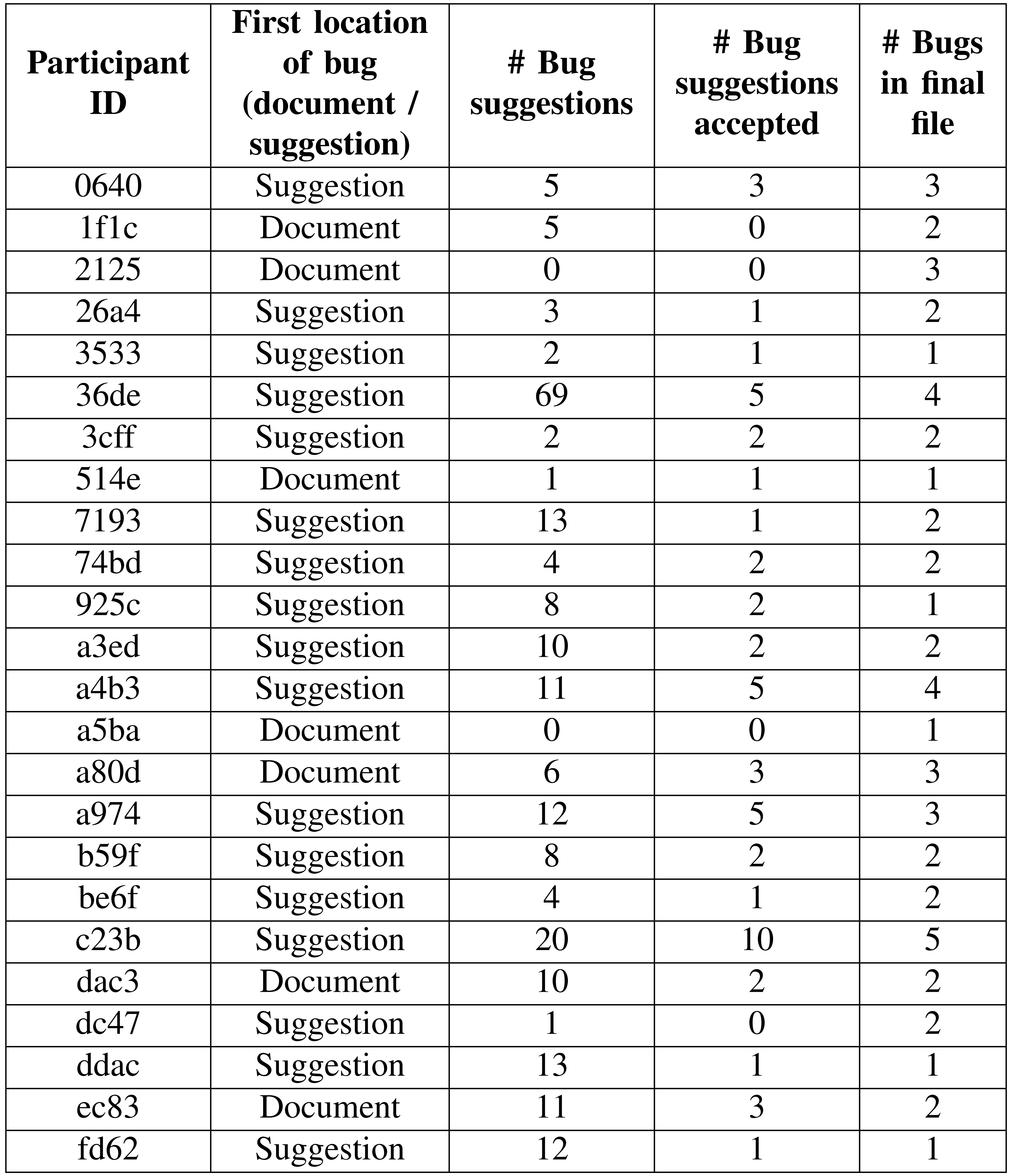
- Using the data from the IDE, can we identify where vulnerabilities were introduced into the user's code?
  - In particular, did they come from **Codex suggestions** or were they written by **humans**?
- **Idea**:
  - Find an automated way to check for some common vulnerability
  - Use our document snapshots and suggestion data to see if it first appeared in a **document** (human-written) or **suggestion** (introduced by Codex)

Bug Origins: Missing `strdup`

- We picked one bug for this that we could identify with just a regular expression
  - Vulnerability failing to make a copy of the `item_name` provided by the caller (e.g. using strdup) before storing it in the node
  - Can lead to **CWE-416: Use-After-Free** because the list library has no control over when the user-provided string will be freed
  - We can identify it by just looking for direct assignments to `node->item_name` with no strdup/strcpy/malloc

Bug Origins: Results

- This vulnerability was introduced by Codex more often than not
- But some users introduced it themselves, and did not accept further buggy suggestions
- Some users got a **lot** of buggy suggestions (69 in one case!)
- Weak trend: more bug suggestions => more bugs in final file



Participant ID	First location of bug (document / suggestion)	# Bug suggestions	# Bug suggestions accepted	# Bugs in final file
0640	Suggestion	5	3	3
1f1c	Document	5	0	2
2125	Document	0	0	3
26a4	Suggestion	3	1	2
3533	Suggestion	2	1	1
36de	Suggestion	69	5	4
3cff	Suggestion	2	2	2
514e	Document	1	1	1
7193	Suggestion	13	1	2
74bd	Suggestion	4	2	2
925c	Suggestion	8	2	1
a3ed	Suggestion	10	2	2
a4b3	Suggestion	11	5	4
a5ba	Document	0	0	1
a80d	Document	6	3	3
a974	Suggestion	12	5	3
b59f	Suggestion	8	2	2
be6f	Suggestion	4	1	2
c23b	Suggestion	20	10	5
dac3	Document	10	2	2
dc47	Suggestion	1	0	2
ddac	Suggestion	13	1	1
ec83	Document	11	3	2
fd62	Suggestion	12	1	1

Conclusions

Check out the paper! https://arxiv.org/abs/2208.09727
Dataset Visualization: https://moyix.net/~moyix/secret/suggestion_cover.html

- Significant differences in functionality between groups on **functionality**
- Surprisingly, **no discernible difference** on security
  - Limited by small sample size
  - *Maybe* a slight trend in favor of Codex
- Potentially found a signal we can use to distinguish **Copilot/Codex** written code from human-written code (repetition)
  - Has implications for stylometry, confirms that tendency toward repetition may *amplify* the existing vulnerabilities in the code