Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants

The question under study in this paper is whether developers with access to an LLM-based code completion assistants produce less secure code than the code produced by programmers without this access? The LLM completion themselves might contain security vulnerabilities, and some programmers could accept the buggy suggestion. In order to explore deeper into this area, the authors focused on three research questions to be answered through this research paper.

1. Does an AI code assistant help novice users write better code in terms of functionality?
2. Given functional benefits, does the code that users write with AI assistance have acceptable incidence rate of security bugs vis-a-vis code written without assistance?
3. How do AI assisted users interact with potentially vulnerable code suggestions—i.e., where do bugs originate in an LLM-assisted system?

The major security concerns that could lead to the LLM outputting vulnerable code suggestions is that it can potentially be trained over insecure or buggy code, the code even though secure in isolation could be insecure based on the sequence it is executed in. The presentation also stated a study that found that 40% of Copilot's code contains security related bugs. Github also states that there is a 0.1% chance that the model would suggest the exact same code without any changes from the training data.

Since, code security is the point of this paper, the different methods to evaluate the security of a piece of code also becomes very important. Static analysis is where the code is analyzed statically during compile-time to detect bugs. On the other hand run-time analysis modifies the program at compile-time using debuggers and sanitizers and tries to prevent and catch bugs during execution. Fuzzers is another popular and extremely effective method of code security evaluation where a large amount of wide variety of random inputs are given to the input points in a program to observe if the functionality of the code changes from what is needed or expected. Also, there is manual analysis which is just humans reviewing the code manually to try and find bugs.

For conducting the experiment, the authors used two groups of people, a control group and an assisted group. The control group acts in the traditional and conventional way of coding which includes using resources such as Google, StackOverflow, Documentations, etc. The assisted group on the other hand would be given suggestions by the LLM model along with the resources that is accessible to the Control Group. The task was a programming assignment consisting of implementing 12 functions in C language that developed a shopping list making use of the linked list data structures. The reason why C was chosen is because almost 70% of CWE(Common Weakness Enumeration) assigned by Microsoft was in C language, primarily since the language inherently have a lot of design choices that makes it easier for the programmers to write insecure code. The participants knew C language from their

syllabus requirements and they were given 2 weeks of time to complete the assignment. The assignment also had certain requirements that were meant to increase the chance of insecure code. For example, when implementing the linked list the participants were not allowed to use node pointers but had to use position index, the linked list had to be one-index based instead of zero-index based and the functions instead of returning the values should return a success/failure state and should write the conventional return value should be set in argument pointers. The functions also had comments that described the functionality of the code along with examples at times. The code suggestion system would collect all the code and prior to the point of writing whenever the user pauses for 750 ms, pass it to code-cushman-001 LLM model and return the suggestion. The user had the choice to accept, reject or modify the suggestion.

The approach had its share of shortcomings and threats to validity that the presenter rightly pointed out. First of all the participants are from the same demographic (college students) and hence need not be reflective of the professional coding world. Adding on to this, the bug count was higher which the authors attributed to time constraints. Also, C language is a fairly hard language to code efficiently in for an inexperienced user and hence the results might be biased towards this. Furthermore, the authors did not capture continuous data from the users but instead relied on screenshots taken every 60 seconds.

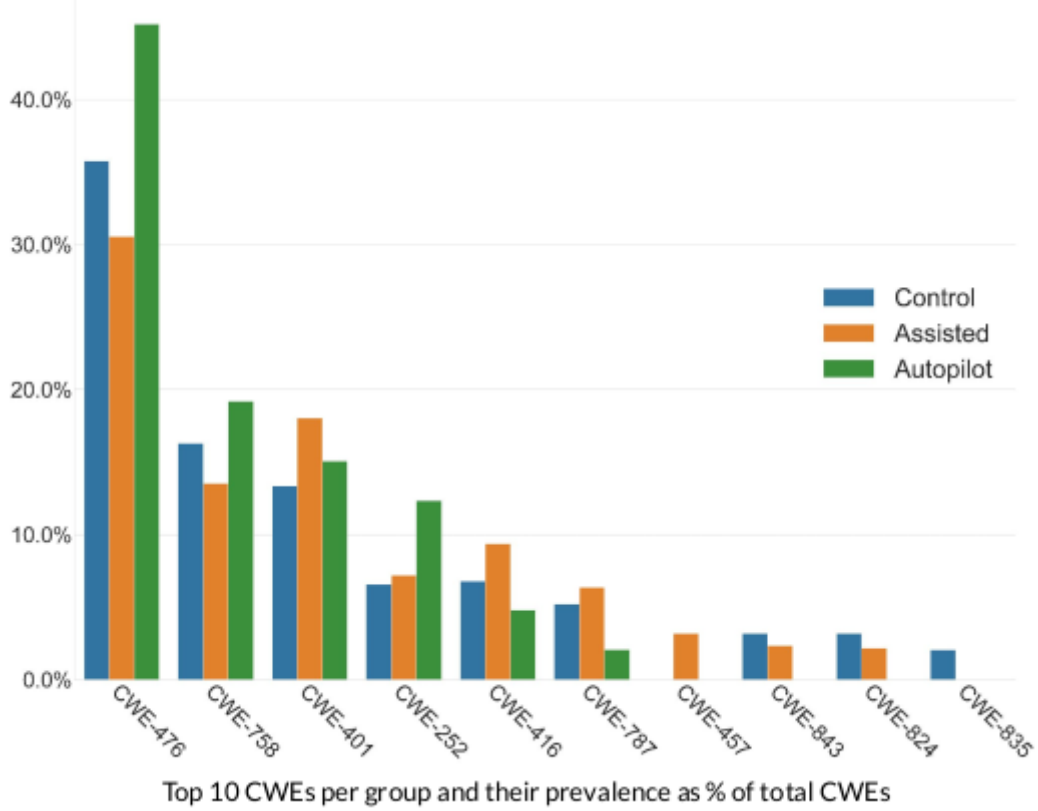
Apart form the 2 user groups who did the assignment, the authors also created 30 solutions entirely generated using LLM, ten each from code-cushman-001, code-davinci-001, and code-davinci-002 (Autopilot). The solutions were created by providing the function definition and previously generated code for the LLM model to analyze and output the new code. The functions were generated until they compile for up to 10 retries.

The major point of evaluation for the paper is to determine if the code generated by LLMs would improve the code quality without decreasing security.

The authors conducted functionality based unit tests which consisted of 11 basic tests and 43 expanded edge case tests. The results showed that there was a consistent but small advantage in favor of the assisted group in comparison to the control group. On the other hand, an interesting observation is that autopilot performed slightly worse than assisted in basic tests whereas it was the other way for the expanded tests. It was also noticed that the assisted group wrote more lines of code than the control group. This fact shows that the assisted users had a higher level of productivity.

In order to test the security of the code, the authors used static analysis tools like Github CodeQL and other strategies such as fuzzing and manual analysis. The quantity/metric used for the purpose of this evaluation was bugs per line of code (LoC). Across the experiments conducted it was noticed that the assisted group had fewer bugs compared to the control group. This observations suggests that the security concerns with LLM assistants might be overhyped and might not be as bad as people expect it to be. The incidence of percentage also differs from one vulnerability to another and that can be

observed from the graph given below



Top 10 CWEs per group and their prevalence as % of total CWEs

Another thread of evaluation was on whether the bugs that do exist in the final code was introduced by the human or suggested by the LLM Model. It was noted that the rate at which vulnerabilities are introduced by the LLM model is very similar to the rate at which humans introduce insecure vulnerable code.

Discussion

- The paper in general sheds a positive light on the whole domain of LLM based code suggestion models and shows that it actually benefits the user in terms of both functionality and security.
- Another point that was emphasized and discussed right from the beginning of the presentation is that there is no actual measure for an acceptable number of bugs. Furthermore, having zero bugs is not really the priority or the goal of the paper as this is impractical. The goal of the paper is for the LLM models to be compared against the existing technologies and tools in this domain to ensure that it doesn't do more harm and improves productivity and/or security. If the model generates lesser bugs than humans then it can be considered as a valuable addition. This is the same philosophy followed in fields like autonomous vehicle were the goal is not to have zero accidents but to have lesser accidents than humans.
- Another question is whether an increase in efficiency and productivity would justify a slight increase in the number of bugs/vulnerabilities? Even if it might justify, the issue there would be to find out the right balance between the both and setting the correct thresholds to achieve this.
- The paper uses bugs per Lines of Code as the metric but it is not really the ideal metric as lines of code doesn't necessarily translate to complexity. Moreover it was observed that Assisted coding produced more lines of code than the control group which would immediately indicate a drop in the

bugs per Lines of Code even though both programs were achieving the same goal. A different metric would be needed to determine the density of bugs in a program.

- Bugs and vulnerabilities were used interchangeably in the presentation and hence the difference between these terms were discussed. Bugs are faults that causes the program to function in ways that was not expected or intended. Vulnerabilities on the other hand are bugs that can be exploited specifically with malicious intentions.
- Another interesting terminology was Trusted codebase. Every code that is built by a programmer is on top of certain assumptions, specifically on certain programs to function properly without the need of consideration from the programmer. For example, trusting that the operating system, the libraries you use, etc is going to function the way it is intended to perform.
- The methodology for this paper is observably different from the ones that were presented in class in the beginning. This is mainly because humans are involved in the research study, with the field being dependent on the prompts and comments provided to the models. In general, research in the field of LLM would involve humans which is unlike most other fields in Computer Science (except for fields like Human Interaction Research).
- Adding on to the point above, it can also be seen that the narrative around LLM models are changing with the initial opinion being negative whereas nowadays it is being shown in a more positive light. This can be again attributed to the human involvement in the field, with humans getting progressively better at using LLM models in a more productive way and achieve better results.
- Another point of discussion was fuzzers and their place in security testing. Fuzzing is where input is randomized (based on certain guidelines and strategies) and fed through the inputs to a program. This can be in the form of command line arguments, file reads, etc. However there can also be implicit inputs that a program's function depends on such as the environment the program is in, the operating system, etc. In the field of autonomous programs like malware it is this kind of fuzzing that becomes more important where the versions of operating system, the system language, region etc are the implicit inputs that are fuzzed.