Subject : Data modeling and source code generation and ai

First article : [Vulnerabilities in AI Code Generators:Exploring Targeted Data Poisoning Attacks (arxiv.org)]

Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks

The article discusses vulnerabilities in AI code generators, particularly focusing on targeted data poisoning attacks. Large language models used for code-related tasks rely on vast amounts of training data collected from online repositories like GitHub. However, the accessibility of this data allows adversaries to tamper with the model's training by injecting malicious code. The attacker can manipulate training data to create poisoned samples, where the code descriptions remain unchanged, but the safe code snippets are replaced with semantically equivalent insecure implementations. The attacker identifies target vulnerabilities commonly found in Python programs and crafts poisoned samples accordingly. They construct a dataset called PoisonPy, containing safe and unsafe code snippets for training AI models. The poisoning process biases the trained model to generate vulnerable code snippets when presented with similar code descriptions during inference. The article proposes a methodology for assessing the vulnerability of AI code generators to targeted data poisoning attacks using NMT models. Three NMT models, including Seq2Seq, CodeBERT, and CodeT5+, are evaluated to understand their susceptibility to such attacks.

- Poisoned Samples : This article discusses also the concept of poisoned samples in the context of AI code generation. A clean training sample consists of a code description (in NL) and its corresponding code snippet. A poisoned sample is crafted by replacing the safe code snippet with an equivalent but insecure implementation, while keeping the code description intact. The attacker selects target vulnerabilities commonly found in Python programs and manipulates training data to create poisoned samples. A dataset named PoisonPy is constructed, containing both safe and unsafe code snippets for training AI models. The poisoning process biases the trained model to generate vulnerable code snippets when presented with similar code descriptions during inference. It proposes a methodology for evaluating the vulnerability of AI code generators to targeted data poisoning attacks using NMT models. Three NMT models, including Seq2Seq, CodeBERT, and CodeT5+, are assessed for their susceptibility to such attacks. These provide an overview of the main points covered in both articles regarding vulnerabilities in AI code generators and the exploitation of poisoned samples for targeted data poisoning attacks.

At the end, there is a study where results are shown : The results indicated that AI code generators are indeed vulnerable to data poisoning attacks. Injecting even a small percentage (around 3%) of poisoned data into the training set resulted in a significant increase in the generation of vulnerable code snippets. Pre-trained models like CodeBERT and CodeT5+ were particularly susceptible to data poisoning, with success rates ranging from approximately 12% to 41%. The success of the attack varied across different vulnerability groups, with data protection issues being the easiest to inject and resulting in the highest success rates. Overall, the experiments highlighted the effectiveness and potential dangers of data poisoning attacks on AI code generators, emphasizing the need for robust defense mechanisms against such attacks.

Second article : [Programming Is Hard – Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation (acm.org)]

The recent emergence of AI-driven code generation tools, such as OpenAI Codex, DeepMind AlphaCode, and Amazon CodeWhisperer, has given discussions about the impact of these tools on programming education. In general, introductory programming courses have faced challenges in teaching students to code, with programming often seen as difficult. However, the availability of AI code generation tools has made writing code for assignments and exams much easier, potentially changing the landscape of computing education. As these tools become more prevalent, it brings up questions about how introductory computing courses will evolve. With students able to easily auto-generate code solutions by inputting problem descriptions into AI-powered tools, the delivery of computing curricula may need to adapt. This shift has the potential to significantly change how programming is taught and learned. In the background and context section, the authors discuss how viable AI-driven code generation has only recently become available to the general public, starting around 2021. These tools claim to make programming more productive and accessible but also present various challenges that need to be addressed. In September 2021, the New York Times published an article introducing OpenAI's Codex model, a descendant of the advanced natural language model GPT-3. Codex is trained on over 50 million GitHub repositories, with a focus on Python, and can generate code in multiple languages, with Python being its strongest. It can also translate code between languages, explain code functionality in English, and return code time complexity. Codex powers GitHub Copilot, which assists developers by generating code suggestions based on natural language descriptions. The model's capabilities have been demonstrated in solving programming tasks, showing promising results in functional correctness and performance. DeepMind also released AlphaCode, a model trained on a vast amount of GitHub code and fine-tuned on competitive programming problems. AlphaCode performs competitively in solving complex programming tasks, demonstrating its ability to generate original solutions based on problem descriptions. Amazon CodeWhisperer, another AI-driven code generation tool, provides code recommendations based on developer comments and prior code. It aims to improve developer productivity by suggesting appropriate cloud services and libraries for given tasks. These AI code generation tools present both opportunities and challenges for students and educators in introductory programming courses. While they offer the potential to enhance learning efficiency and productivity, they also raise concerns about reliance on automated solutions and the need for educators to adapt teaching practices accordingly. It is essential for educators to review their approaches to accommodate these new technologies and ensure that students develop a deep understanding of programming concepts despite the availability of AI-generated code solutions.

The emergence of AI-powered code generation tools presents various opportunities and challenges in the field of education, much like handheld calculators did in mathematics education. These tools offer opportunities for learning and teaching in several ways:

There is :

- exemplar Solutions an AI-generated solutions provide students with exemplar solutions to programming problems, aiding them in checking their work and understanding different approaches.

-Variety of Solutions, these tools expose students to the diversity of problem-solving approaches, helping them appreciate different coding styles and strategies.

-Code Review of Solutions: an Assessment can focus on evaluating the quality and style of solutions rather than just correctness, encouraging discussions around code quality and refactoring.

-Exercise Generation: AI models can generate programming exercises and code explanations, offering novel learning resources that can enhance understanding and engagement.

-Code Explanations: High-quality explanations of code generated by AI models can help learners develop a robust understanding of programming concepts.

-Illustrative Examples: AI-generated examples can illustrate various programming constructs and algorithms, providing students with more effective learning resources.

-Explaining Algorithmic Concepts Clearly: AI models can assist in clarifying algorithmic concepts by generating solutions based on simplified problem descriptions, helping students communicate algorithmic problems more effectively.

-Alleviating Programmer's Writer's Block: These tools can help students get started with programming assignments by generating starter code, facilitating forward momentum in learning.

-Overcoming Traditional Barriers: AI-generated code explanations can help mitigate common barriers faced by novice programmers, such as compiler error messages, improving the learning experience.

However, along with these opportunities, there are also challenges associated with the use of AI-generated code tools:

-Academic Misconduct: There is a risk of students using AI-generated code to complete assignments without fully understanding the concepts, leading to academic misconduct and undermining the learning process. Attribution: Determining the ownership and attribution of AI-generated code poses challenges, especially when distinguishing between human and machine-generated contributions. AI models may be trained on public code that is not suitable for novice students, potentially introducing biases and complexities beyond their level of understanding. There is harmful Biases that also means that AI-generated code may reflect stereotypes and biases present in the training data, raising concerns about the appropriateness and inclusivity of the generated solutions and the topic of the security of AI-generated code is crucial, as it may introduce vulnerabilities if not properly scrutinized and validated. Addressing these challenges requires careful consideration and adaptation of educational practices to ensure that AI-generated code tools are used effectively to enhance learning outcomes without compromising academic integrity or perpetuating harmful biases.

In conclusion, AI-generated code is rapidly becoming an integral part of programming education, yet educators are still grappling with how to effectively navigate its challenges and leverage its benefits. As the prevalence of auto-generated code increases in software development, it is evident that a shift towards emphasizing code reading and evaluation is necessary. This aligns with instructional theories advocating for a deeper understanding of code rather than solely focusing on its generation. Moreover, the widespread adoption of AI-generated code necessitates a thorough examination of its ethical implications. Educators must guide students through ethical reflection, preparing them to navigate the complex ethical landscape of the tech industry. Titus Winters, a Principal Software Engineer at Google, highlighted the importance of ethical awareness in programming, emphasizing its significance alongside technical proficiency. As AI-generated code becomes more prevalent in industry practices, educators must proactively address ethical issues in computing education from the outset of the curriculum. Failure to do so risks falling behind in shaping the future of programming education and addressing enduring challenges in a rapidly evolving landscape. Therefore, concerted efforts are needed to adapt pedagogical approaches and foster ethical awareness among students to effectively navigate the evolving landscape of AI-generated code in programming education.

Learning to write code with the help of computers is getting more common. But we need to think about whether it's fair and right to use these tools. We also need to focus on understanding the code they make for us. Being a good person is just as important as being good at writing code. If we don't teach students about these things, we might miss out on helping them learn the best way.

Third : Software Heritage: Why and How to Preserve Software Source Code - Archive ouverte HAL

Software is omnipresent, serving as the engine driving innovation across industries, shaping our communication, entertainment, trade, and even political landscapes. It acts as the conduit through which we access digital information, underpinning modern scientific research in every field. In essence, software constitutes an expanding reservoir of cultural, scientific, and technical knowledge. However, the essence of software knowledge doesn't reside in its executable form but rather in its human-readable source code—the preferred medium for developers to understand and modify programs. Unlike optimized binaries, source code is designed for human comprehension while remaining readily convertible into machine-executable format. Source code holds a unique position as a form of knowledge that bridges the human and machine realms. It evolves to accommodate new needs and contexts, necessitating access to its complete development history for comprehensive understanding. The advent of Free/Open Source Software (FOSS) has propelled the creation of a software commons—a repository of widely available, reusable, and modifiable software. Despite its significance, source code preservation has not received adequate attention.

Several factors contribute to the neglect of source code preservation:

-Source Code Diaspora: With millions of projects hosted on various platforms, including GitHub, GitLab, and SourceForge, source code is dispersed across different repositories, making it challenging to track and manage. A centralized repository is needed to consolidate all publicly available source code, akin to a modern "great library" of source code.

-Fragility of Source Code: Digital data, including source code, is susceptible to loss due to human error, hardware failure, hacking, or platform shutdowns. While code hosting platforms facilitate collaboration, they do not guarantee long-term preservation. The abrupt shutdown of platforms like Gitorious and Google Code underscores the need for dedicated preservation efforts.

-Lack of a Comprehensive Archive: Despite the critical role of source code in advancing science and industry, there is a dearth of comprehensive archives dedicated to source code preservation. A centralized archive is needed to ensure that source code remains accessible even if platforms or repositories disappear.

To address these challenges, initiatives like Software Heritage have emerged to build a comprehensive archive of publicly available source code. By providing a unified platform for accessing and analyzing source code, such initiatives aim to safeguard software knowledge for future generations and facilitate research on a vast scale. Software Heritage is a big project that's finally gathering all the software that's out there, keeping track of how it's been developed, and organizing it neatly. This is a huge job with lots of challenges, from technical stuff to making sure it can keep going in the long run. Software Heritage is trying to fix a big problem: we're not taking good care of all the computer code that's out there. There are three main reasons why:

Code is scattered everywhere. With so many projects on different platforms like GitHub and others, it's hard to keep track of it all. Plus, projects move around a lot, making it even harder to find. Code is fragile. Digital information can easily get lost or damaged, and many code hosting platforms don't guarantee that your code will be safe forever. Some platforms have even shut down, causing lots of projects to scramble for a new home. We need better tools to understand all the code. With so much software out there, we need a way to analyze it all and learn from it. But right now, we don't have a good way to do that. To tackle these problems, Software Heritage was started in 2016. Its goal is to collect, organize, and save all the public code, no matter where it's from or how it's shared. But it's a big job with lots of challenges, like figuring out where all the code is and how to keep it safe. Computer scientists are working on it, but it's going to take time and effort to get it right

So, to make sure it succeeds, Software Heritage follows some key principles. It keeps things simple and avoids duplication, and it's open to everyone to get involved. It's not just about the code itself but also about making sure everyone can access it and use it. Software Heritage is open for anyone to join in and help archive all the software we have. It's not easy, but it's really important.

Fourth : Microsoft Word - aaai2004.doc (ualberta.ca)

Code Generation for AI Scripting in Computer Role-Playing Games

In the world of computer role-playing games (CRPGs), players crave immersive experiences filled with rich characters and captivating stories. To meet this demand, game developers provide tools like the Aurora toolset, allowing players to create their own game content. In computer role-playing games (CRPGs) like Neverwinter Nights, players want more than just fancy graphics – they want smarter characters and better stories. To meet this demand, game developers include tools like the Aurora toolset, which lets players create their own game content. However, many players, especially those without programming experience, found the scripting system in Aurora too complex. That's where ScriptEase comes in. ScriptEase is a tool we created to make scripting in CRPGs easier for everyone, from beginners to experts. With ScriptEase, even people who aren't programmers can create their own game content without getting lost in the technical details. We designed it to be simple enough for beginners to use, but still powerful enough to satisfy the needs of experienced programmers. One of the biggest challenges in creating ScriptEase was making it user-friendly while still allowing for complex scripting. We analyzed thousands of scripts from Neverwinter Nights to understand what users needed and how they approached scripting. We found that many users struggled with basic programming concepts like variables and conditionals. So, we designed ScriptEase to handle these concepts in a way that's intuitive for beginners. ScriptEase also helps organize and test scripts, making it easier for users to manage their game content. We believe that by making scripting more accessible, we can empower players to unleash their creativity and make even more amazing games." This version provides a bit more detail while still keeping the language simple and easy to understand. Let me know if you need further adjustments.

In summary, this paper has introduced ScriptEase, a scripting tool tailored for computer role-playing games (CRPGs). While ScriptEase is not intended for serious systems programming, its template-based script generation model has proven effective for knowledge-intensive automation tasks. Through engagement with our user community, we have gained valuable insights into areas for improvement in ScriptEase's design.

Driven by the needs of non-programmers, ScriptEase has focused on simplifying scripting tasks and empowering users to create believable behaviors for NPCs within CRPGs. However, there remains a significant demand for NPCs with behaviors that can interact seamlessly with plot constraints. As such, ScriptEase is evolving to meet this need, with a particular emphasis on implementing behavior patterns that optimize CPU and memory usage while maintaining uniqueness and interest.

Moving forward, the development of ScriptEase will continue to prioritize user feedback and the ongoing refinement of its design to better serve the needs of CRPG enthusiasts. By addressing the challenges of creating believable NPC behaviors, ScriptEase aims to enhance the immersive gaming experience and contribute to the advancement of virtual-world inhabitants in CRPG environments.