



**black hat**<sup>®</sup>  
USA 2024

**AUGUST 7-8, 2024**  
BRIEFINGS

# **From HAL to HALT: Thwarting Skynet's Siblings in the GenAI Coding Era**

Chris Wysopal

Co-founder & CTO, Veracode

**VERACODE**

#BHUSA @BlackHatEvents

One of the 1<sup>st</sup> vulnerability researchers, member of hacker think tank, L0pht in 1990s



Unites States Senate testimony - 19 May 1998



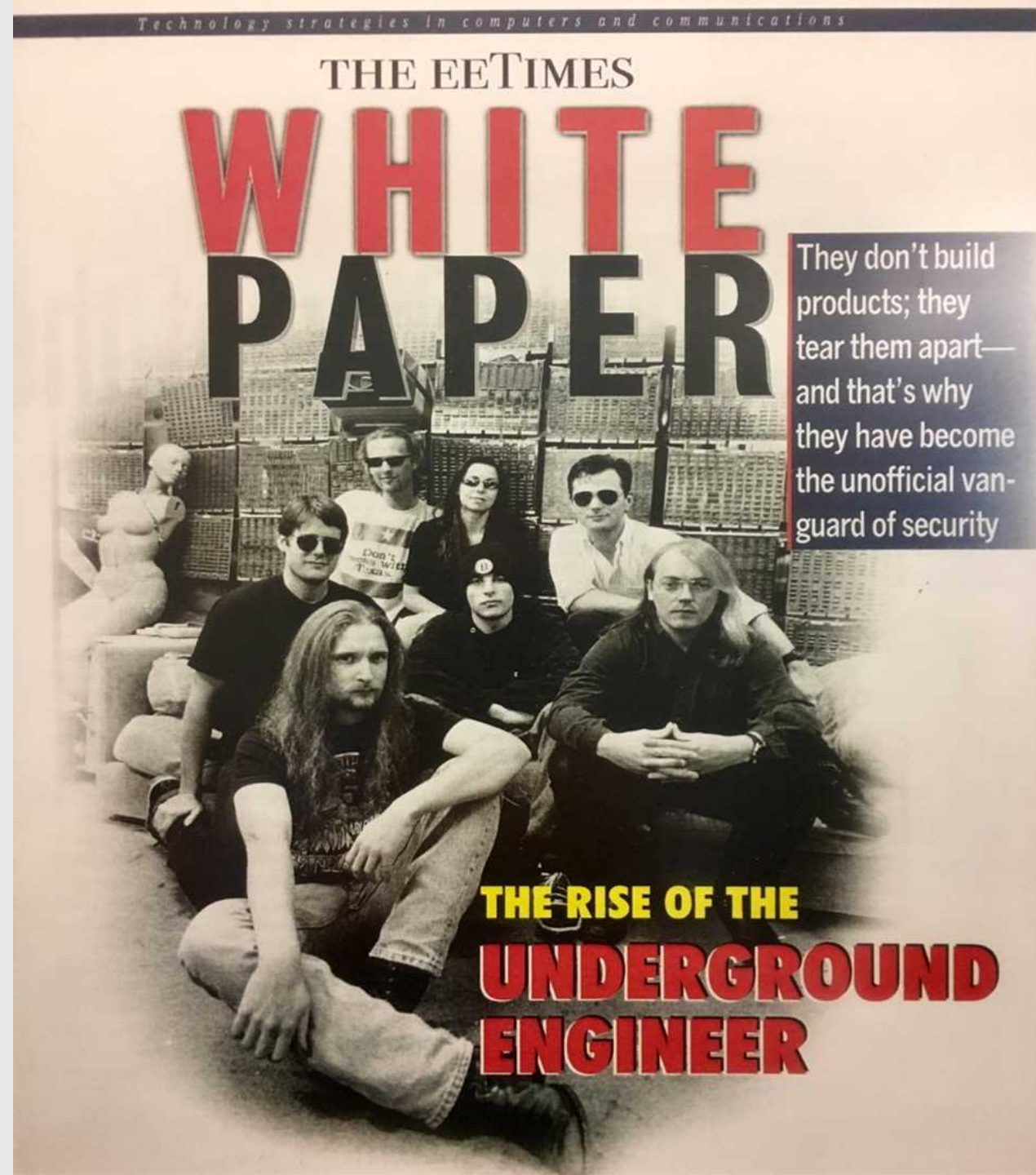
**Into the light:** *Once shadowy computer code warriors like Kingpin are going legit*

## Using Good Hackers to Battle Bad Hackers

**I**F YOU HAVE A MURKY PAST AND DOUBT you could become a dot-com millionaire, think again. Last week a scraggly band of hackers known as “LOpht Heavy Industries” joined with some straitlaced tech execs to form @Stake, an Internet-security consulting firm.

Newsweek, January 17, 2000

Improve the  
Security of  
Your *Product*  
by Breaking  
Into It





**Founded  
@stake security  
research team  
and then  
Veracode to  
build security  
into SDLC**

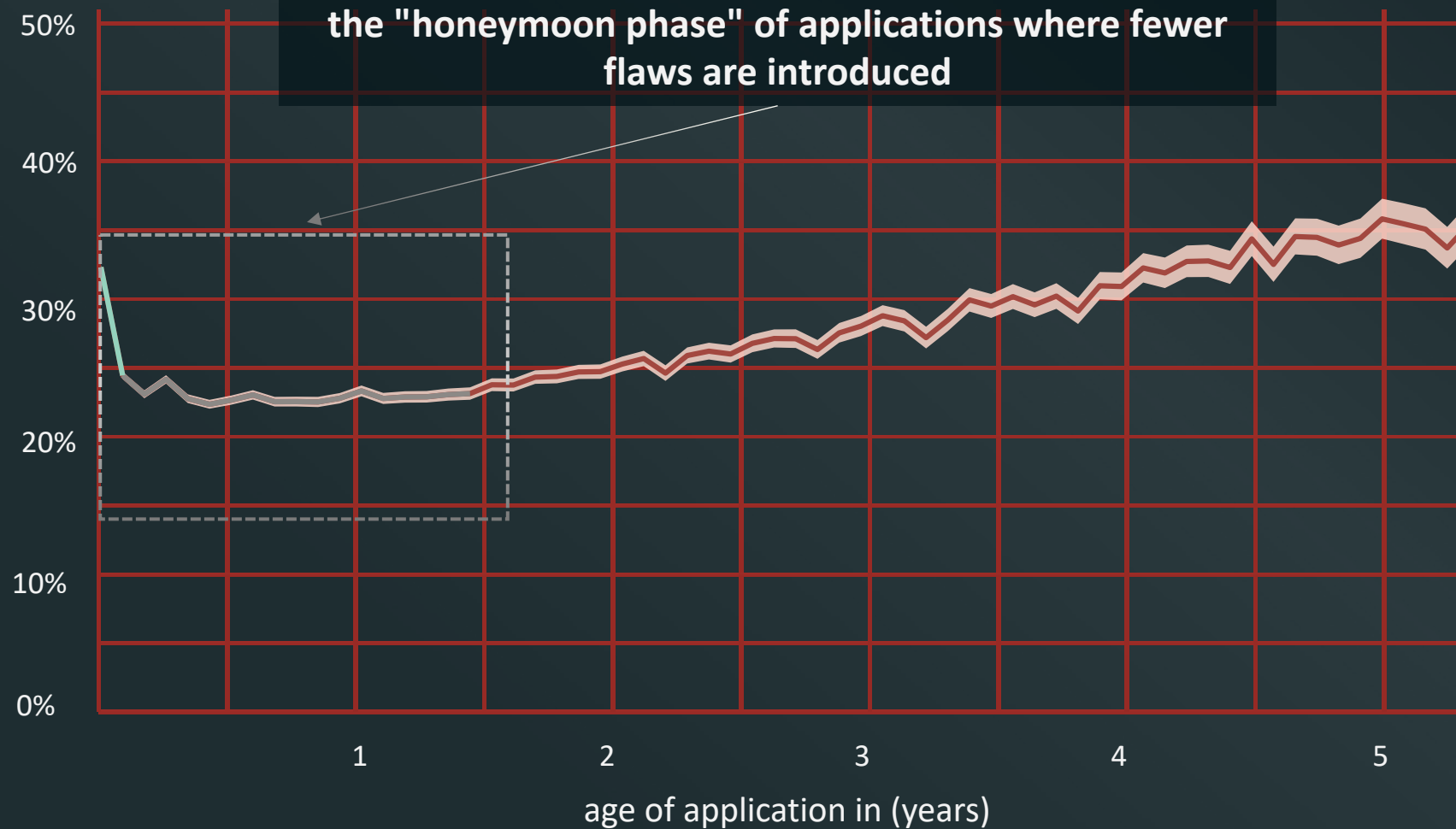


A silhouette of a person sitting cross-legged on a dark, jagged mountain peak. The person is holding a glowing blue laptop. The mountain and the background are filled with binary code (0s and 1s) in various colors (blue, red, white). The background is a light blue gradient.

# State of Software Security 2024

Addressing the  
Threat of Security Debt

# new flaws introduced by application age





# organizations are drowning in **security debt**



**70.8%**  
of organizations  
have security  
debt



**45%**  
of organizations  
have critical  
security debt





**few teams  
fix flaws fast  
enough** to reduce  
security risk at a  
**meaningful pace**



## why software security is **hard**

- security knowledge gaps
- increased application complexity
- incomplete view of risk
- evolving threat landscape



Let's add the exciting potential of large language models that can write code!







# Developer GenAI use right now

## Generating code

Understanding code/Code review

Remediating defects

Translating programming languages

Creating and maintaining unit tests

Writing documentation

# Emerging dev uses for GenAI

Learning about the code base

Searching for answers to avoid  
reinventing the wheel

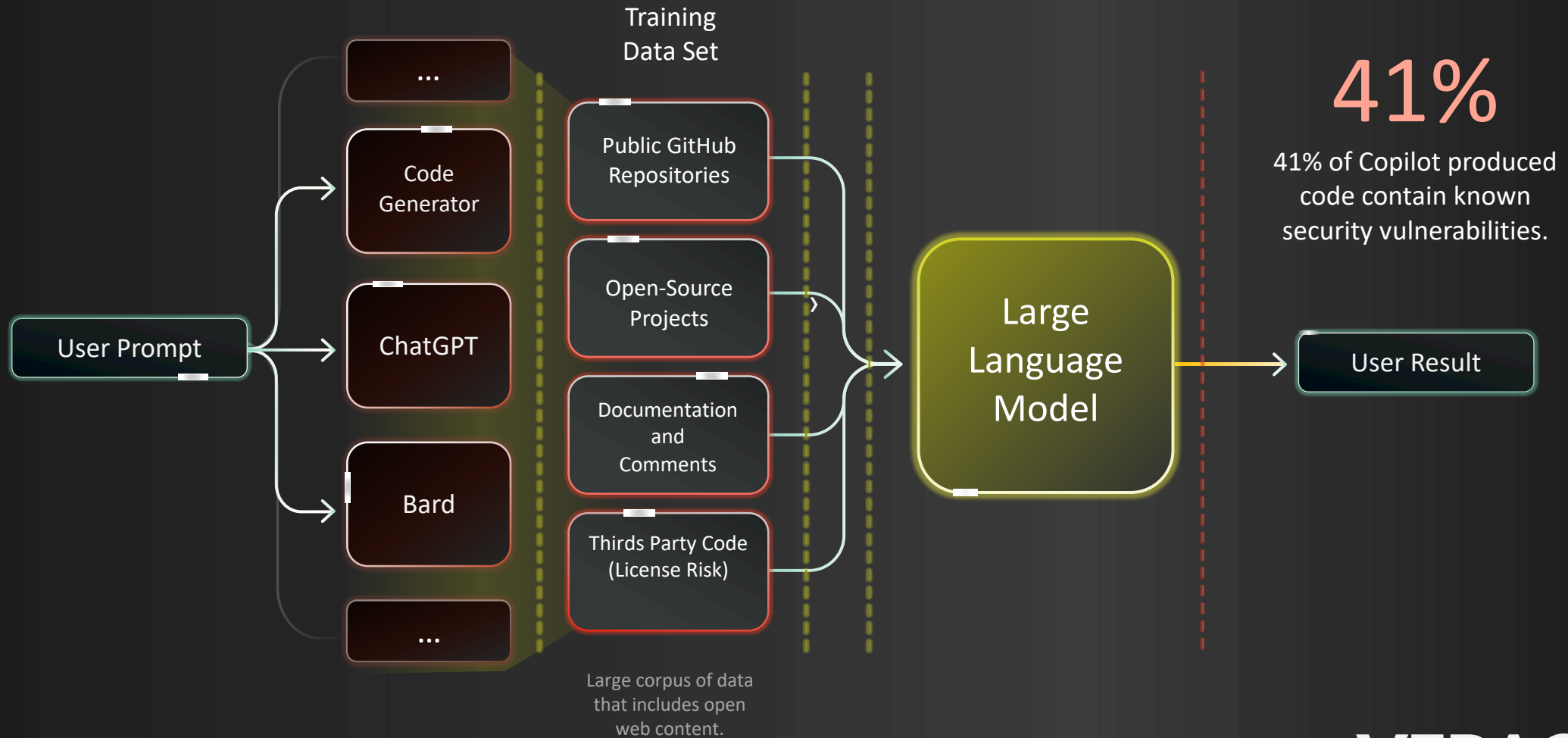
Reading log files to find a root  
cause

Creating and running  
functional & non-functional  
tests

Remediating security  
vulnerabilities



# Large Language Models





# Security implications of LLMs

Wuhan University Study on AI Code Generators

New York University Study on GitHub Copilot

Stanford University Study on AI Code Generators

Purdue University Study on ChatGPT and AI

36%

Out of the 435 Copilot generated code snippets found in repos 36% contain security weaknesses across 6 programming languages.

41%

Of 1689 generated programs 41% of Copilot produced programs contained vulnerabilities

52%

Developers using LLMs were more likely to write insecure code.

They were more confident their code was secure.

52% of ChatGPTs answers were incorrect. Developers preferred them 35% of the time yet 77% of those answers were wrong

## Security Weaknesses of Copilot Generated Code in GitHub

Yujia Fu School of Computer Science, Wuhan University, Wuhan, China  
Peng Liang School of Mathematical and Computational Sciences, Anhui University, Maanshan, China  
Mojtaba Shahin School of Computer Science, BMIT University, Melbourne, Australia  
Jian Yu School of Computer Science, Wuhan University, Wuhan, China

**ABSTRACT** Modern code generation tools use AI models, particularly Large Language Models (LLMs), to generate functional and complete code. While such tools are becoming popular and widely available for developers, using these tools is often accompanied by security challenges, leading to increased bugs and errors in the code. Therefore, it is important to assess the quality of the generated code, especially in terms of its security. Researchers have recently explored various aspects of code generation tools, including security. However, many open questions about the security of the generated code require further investigation, especially the security of automatically generated code in the wild. In this work, we conducted an empirical study by analyzing the security weaknesses in code snippets generated by GitHub Copilot. We found a total of 1689 security weaknesses in Copilot generated code. The goal is to investigate the types of security issues and their risk to the world ecosystem further than crafted scenarios. To this end, we identify that 61% code snippets generated by GitHub Copilot from publicly available projects that crafted scenarios. To this end, we identify that 61% code snippets generated by GitHub Copilot from publicly available projects, which can include test descriptions (comments), code (such as function arguments, expressions, variable names, etc.) or a combination of both and code [34]. After removing these code snippets, the results show that (1) 36.8% of Copilot generated code snippets contain CVEs, and these errors are generated across multiple languages; (2) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code; (3) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code; (4) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code.

### 1 INTRODUCTION

Code generation tools aim to automatically generate functional code based on prompts, which can include test descriptions (comments), code (such as function arguments, expressions, variable names, etc.) or a combination of both and code [34]. After removing these code snippets, the results show that (1) 36.8% of Copilot generated code snippets contain CVEs, and these errors are generated across multiple languages; (2) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code; (3) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code; (4) the security weaknesses are detected in the generated code, which, in turn, can be used to generate code.

## Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions

Hammond Pearce School of Computer Science, Brock University, St. Catharines, ON, Canada  
Balogh Ahmad Department of ECE, Brock University, St. Catharines, ON, Canada  
Benjamin Tan Department of ECE, Brock University, St. Catharines, ON, Canada  
Brendan Dolan-Gavitt Department of ECE, Brock University, St. Catharines, ON, Canada  
Ramesh Karri Department of ECE, Brock University, St. Catharines, ON, Canada

**ABSTRACT** There is burgeoning interest in deploying AI-based systems to assist humans in developing computer systems, including tools that automatically generate complete code. The most notable of these systems is the form of the so-called "AI code generator" GitHub Copilot, which is a language model trained on open-source code. However, such tools often contain bugs—and so, given the vast quantity of unreviewed code that Copilot has generated, it is vital that the language model will not have learned from vulnerabilities, bugs, etc. This paper examines the security of Copilot's code contributions. In this work, we systematically investigate the prevalence and conditions that cause GitHub Copilot to recommend insecure code, and we examine the security of Copilot's code contributions. In this work, we systematically investigate the prevalence and conditions that cause GitHub Copilot to recommend insecure code, and we examine the security of Copilot's code contributions. In this work, we systematically investigate the prevalence and conditions that cause GitHub Copilot to recommend insecure code, and we examine the security of Copilot's code contributions.

**INTRODUCTION** With increasing pressure on software developers to produce code quickly, there is considerable interest in tools and techniques for improving productivity. The most recent examples of these tools are machine learning (ML)-based code generation, in which large models originally designed for natural language processing (NLP) are trained on past code to generate code. In this work, we investigate the security of Copilot's code contributions. In this work, we systematically investigate the prevalence and conditions that cause GitHub Copilot to recommend insecure code, and we examine the security of Copilot's code contributions.

## Do Users Write More Insecure Code with AI Assistants?

Neil Perry\* School of Computer Science, Stanford University, Stanford, CA, USA  
Megha Srivastava\* School of Computer Science, Stanford University, Stanford, CA, USA  
Deepak Kumar School of Computer Science, Stanford University, Stanford, CA, USA  
Dai Bomb\* School of Computer Science, Stanford University, Stanford, CA, USA

**ABSTRACT** AI code assistants have emerged as powerful tools that can aid in the software development lifecycle and can improve developer productivity. Unfortunately, such assistants have also been found to produce insecure code or code vulnerabilities, raising significant concerns about their usage in practice. In this paper, we conduct a user study to measure how users interact with AI code assistants to solve a variety of security-related tasks. Overall, we find that participants who had access to AI assistants were significantly less secure than those without access to an assistant. Participants with access to AI assistants were also more likely to believe they were more secure, suggesting that such tools may lead users to be overconfident about security issues in their code. To better inform the design of future AI-based code assistants, we release our user study questions and interpretations from researchers seeking to build on our work in the future.

**KEYWORDS** Programming assistants, human and societal aspects of security and privacy, language models, machine learning, LLMs, AI assistants

**1 INTRODUCTION** AI code assistants, like GitHub Copilot, have emerged as programming tools with the potential to lower the barrier of entry for software development and increase developer productivity [23]. These tools leverage underlying machine learning models, like OpenAI GPT-3 and Facebook's LLaMA [15, 11], that use pre-trained large datasets of publicly available code to generate code. While recent work has demonstrated that such models may eventually produce security weaknesses [17], it is not yet clearly understood the security "barriers" associated with their use.

## Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions

Samia Kabir Purdue University, West Lafayette, USA  
David M. Ure-Ionesh Purdue University, West Lafayette, USA  
Binran Kou Purdue University, West Lafayette, USA  
Tianyi Zhang Purdue University, West Lafayette, USA

**ABSTRACT** Over the last decade, Q&A platforms have played a crucial role in how programmers seek help online. The emergence of ChatGPT, however, is causing a shift in the pattern. Despite ChatGPT's popularity, there has been a thorough investigation into the quality and usability of its responses to software engineering queries. To address this gap, we conducted a comprehensive analysis of ChatGPT's replies to 131 questions from Stack Overflow [20]. We analyzed the correctness, consistency, comprehensiveness, and conciseness of these responses. Additionally, we conducted an extensive linguistic analysis and a user study to gain insights into the linguistic and human aspects of ChatGPT's answers. Our results revealed that 52% of ChatGPT's answers contain inaccuracies and 77% are not verbatim reproductions, even for highly repetitive questions [6, 15, 41]. The ability to engage in interactive conversations and provide solutions using natural language has propelled LLMs into becoming a popular option among programmers.

**KEYWORDS** Software and its engineering, General and reference, Empirical studies, LLMs

**1 INTRODUCTION** Software developers often need to answer questions for a variety of software engineering tasks, e.g., API-related, but finding comprehensive or complete answers is often a challenge. The emergence of ChatGPT, however, is causing a shift in the pattern. Despite ChatGPT's popularity, there has been a thorough investigation into the quality and usability of its responses to software engineering queries. To address this gap, we conducted a comprehensive analysis of ChatGPT's replies to 131 questions from Stack Overflow [20]. We analyzed the correctness, consistency, comprehensiveness, and conciseness of these responses. Additionally, we conducted an extensive linguistic analysis and a user study to gain insights into the linguistic and human aspects of ChatGPT's answers. Our results revealed that 52% of ChatGPT's answers contain inaccuracies and 77% are not verbatim reproductions, even for highly repetitive questions [6, 15, 41]. The ability to engage in interactive conversations and provide solutions using natural language has propelled LLMs into becoming a popular option among programmers.

arXiv:2310.02059v1 [cs.SE] 3 Oct 2023

arXiv:2211.03622v3 [cs.CR] 18 Dec 2021

arXiv:2211.03622v3 [cs.CR] 18 Dec 2023

arXiv:2308.02312v3 [cs.SE] 10 Aug 2023



# SALLM Framework For measuring LLM vulnerability generation - Notre Dame

## VULNERABILITIES FOUND IN THE CHATGPT-GENERATED PYTHON CODES

CWE Name	CWE Top-25 Rank	# Vuln. Samples
<b>CWE-312 Cleartext Storage of Sensitive Information</b>	-	14
<b>CWE-798 Use of Hard-coded Credentials</b>	18	5
<b>CWE-208 Observable Timing Discrepancy</b>	-	3
<b>CWE-215 Insertion of Sensitive Information Into Debugging Code</b>	-	3
<b>CWE-338 Use of Cryptographically Weak Random Generator</b>	-	3
<b>CWE-79 Cross-site Scripting</b>	2	2
<b>CWE-209 Generation of Error Message Containing Sensitive Information</b>	-	2
<b>CWE-287 Improper Authentication</b>	13	1
<b>CWE-295 Improper Certificate Validation</b>	-	1
<b>CWE-918 Server-Side Request Forgery</b>	19	1

Vulnerable@k metric best to worst:

StarCoder  
GPT-4:  
GPT-3.5:  
CodeGen-2.5-7B:  
CodeGen-2B:

### Generate and Pray: Using SALLM to Evaluate the Security of LLM Generated Code

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller  
Department of Computer Science and Engineering,  
University of Notre Dame, Notre Dame, IN USA 46556

**Abstract**—With the growing popularity of Large Language Models (LLMs) in software engineers' daily practices, it is important to ensure that the code generated by these tools is not only functionally correct but also free of vulnerabilities. Although LLMs can help developers to be more productive, prior empirical studies have shown that LLMs can generate insecure code. There are two contributing factors to the insecure code generation. First, existing datasets used to evaluate LLMs do not adequately represent genuine software engineering tasks sensitive to security. Instead, they are often based on competitive programming challenges or classroom-type coding tasks. In real-world applications, the code produced is integrated into larger codebases, introducing potential security risks. Second, existing evaluation metrics primarily focus on the functional correctness of the generated code while ignoring security considerations. Therefore, in this paper, we described SALLM, a framework to benchmark LLMs' abilities to generate secure code systematically. This framework has three major components: a novel dataset of security-centric Python prompts, configurable assessment techniques to evaluate the generated code, and novel metrics to evaluate the models' performance from the perspective of secure code generation.

Although LLM-based code generation techniques may produce functionally correct code, prior works showed that they can also generate code with vulnerabilities and security smells [5]-[8]. A prior study has also demonstrated that training sets commonly used to train and/or fine-tune LLMs contain harmful coding patterns, which leak to the generated code [9]. Moreover, a recent study [6] with 47 participants showed that individuals who used the `codex-davinci-002` LLM wrote code that was *less secure* compared to those who did not use it. Even worse, participants who used the LLM were *more likely to believe that their code was secure*, unlike their peers who did not use the LLM to write code.

There are two major factors contributing to this unsafe code generation. First, code LLMs are evaluated using *benchmarks*, which do not include constructs to evaluate the security of the generated code [10], [11]. Second, existing *evaluation metrics* (e.g., pass@k [12], CodeBLEU [13], etc.) assess models' performance with respect to their ability to produce *functionally* correct code while ignoring security concerns. Therefore, the performance reported for these models overly focuses on improving the precision of the generated code with respect to passing the *functional* test cases of these benchmarks without evaluating the *security* of the produced code.

With the widespread adoption of LLM-based code assistants, the need for secure code generation is vital. Generated code containing vulnerabilities may get unknowingly accepted by developers, affecting the software system's reliability. Thus, to fulfill this need, this paper describes a framework to perform Security Assessment of LLMs (SALLM). Our framework includes a (1) a manually curated dataset of prompts from a variety of sources that represent typical engineers' intent; (2) an automated approach that relies on static and dynamic analysis to automatically evaluate the security of LLM generated Python code; and (3) two novel metrics (`secure@k` and `vulnerability@k`) that measure to what extent an LLM is capable of generating secure code.

The contributions of this paper are:

- A novel framework to *systematically and automatically evaluate the security of LLM generated code*;
- A publicly available dataset of Python prompts<sup>1</sup>;

<sup>1</sup>The dataset will be made public on GitHub upon acceptance.

arXiv:2311.00889v2 [cs.SE] 3 Jun 2024

<https://arxiv.org/abs/2311.00889>



# Implications of LLM code generation

Code reuse goes down

Code velocity goes up

Vulnerability density  
similar

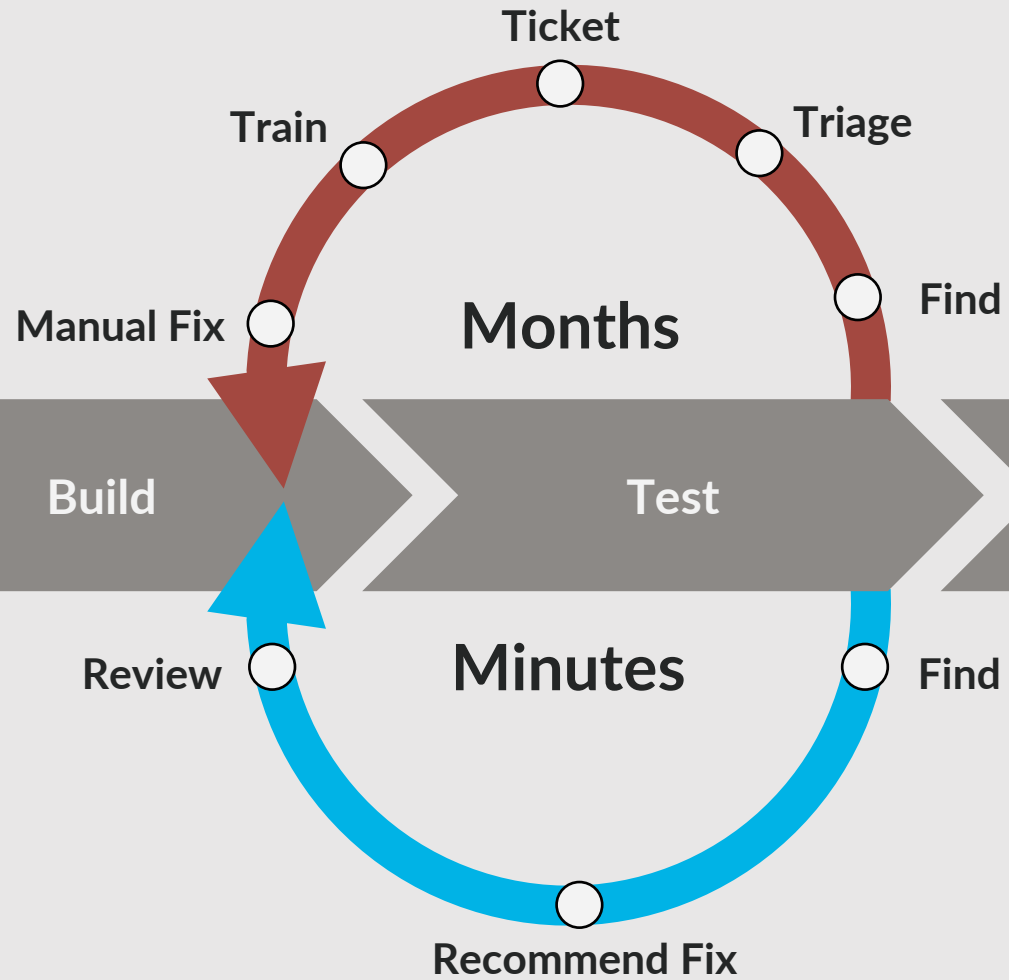
=

Increased Vulnerability  
Velocity



**How can we apply AI to the problem of insecure code, but in a more accurate and trustworthy manner?**

# We need a faster test and fix workflow



# Training data set: Java XSS

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {  
    String name = req.getParameter("name");  
    String[] array = new String[10];  
    array[0] = name;  
    PrintWriter writer = resp.getWriter();  
    writer.println("Hello " + array[0]);  
}
```

← Cross-site scripting (CWE 80)

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {  
    String name = req.getParameter("name");  
    String[] array = new String[10];  
    array[0] = name;  
    PrintWriter writer = resp.getWriter();  
    writer.println("Hello " + StringEscapeUtils.escapeHtml4(array[0]));  
}
```

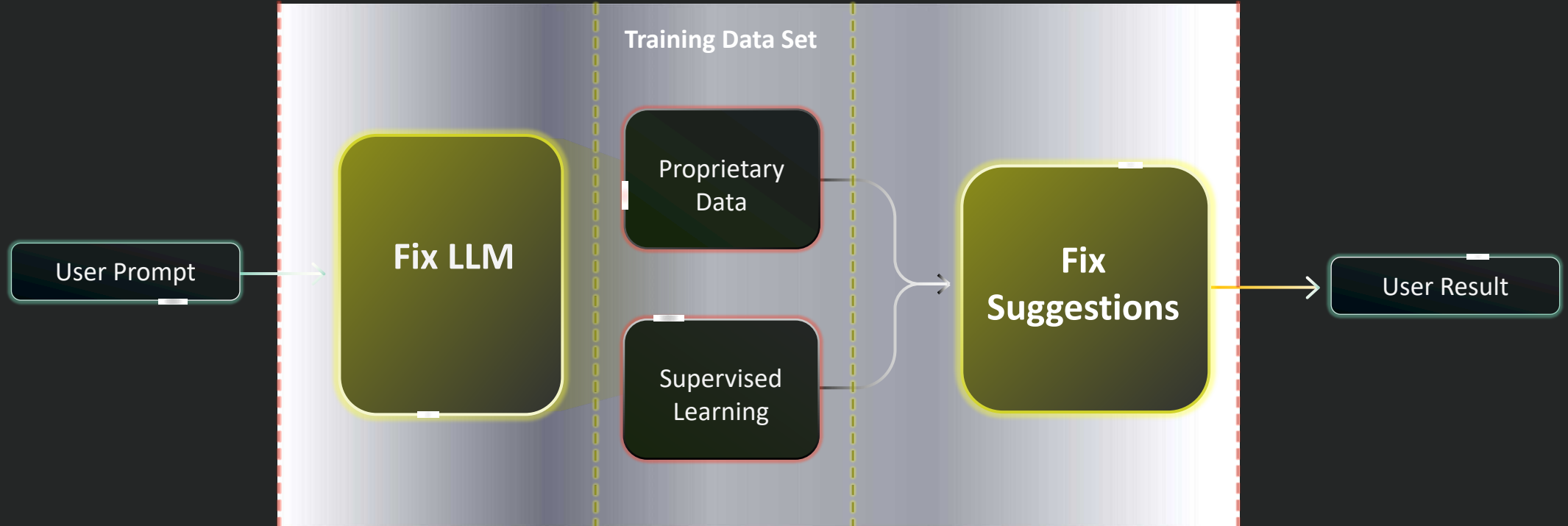


# Fix Approach

Curated Dataset

Code Provenance Assurance

Coverage all that matter

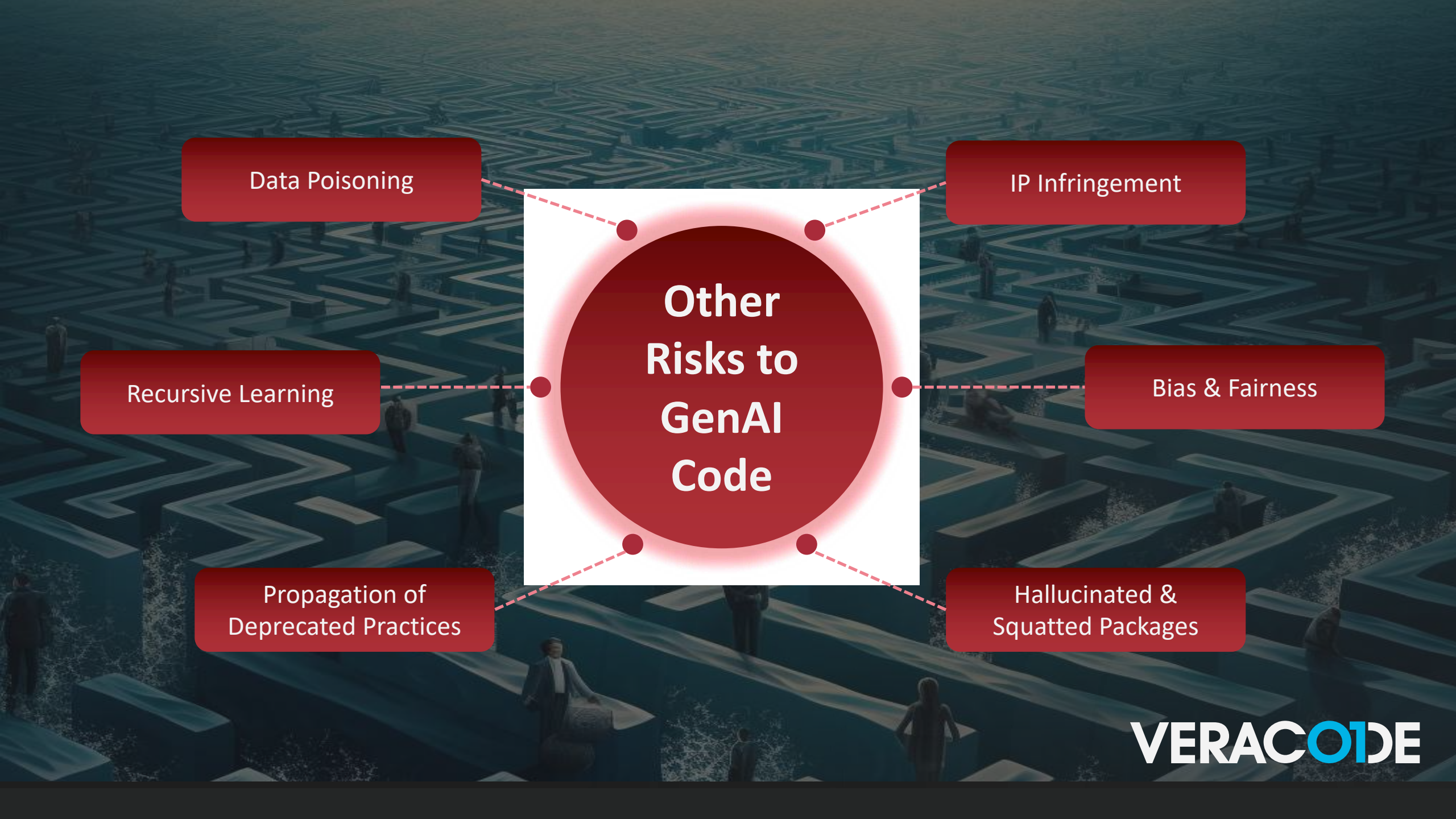


# Recommendations for AI and code security

Consider the implementation details before leveraging AI for developing and/or securing code

- What does the ML model use for training data?
- Is that training data trustworthy/vetted?
- Are there licensing issues with generated code?
- Is any of my intellectual property being leaked?
- How accurate are the generated fixes?

Be aware of human biases that trick us into feeling overly confident about the correctness of AI-generated content



Data Poisoning

IP Infringement

Recursive Learning

Bias & Fairness

Propagation of  
Deprecated Practices

Hallucinated &  
Squatted Packages



GenAI in dev is a powerful tool that requires the **same level of security scrutiny and best practices** as any other aspect of software development

Include security considerations in GenAI prompts

Automate as much of security process as possible, including automated fixing

**Chris Wysopal**  
Co-founder & CTO Veracode  
[@weldpond](#)

**VERACODE**

