black hat 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

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

Unites States Senate testimony - 19 May 1998

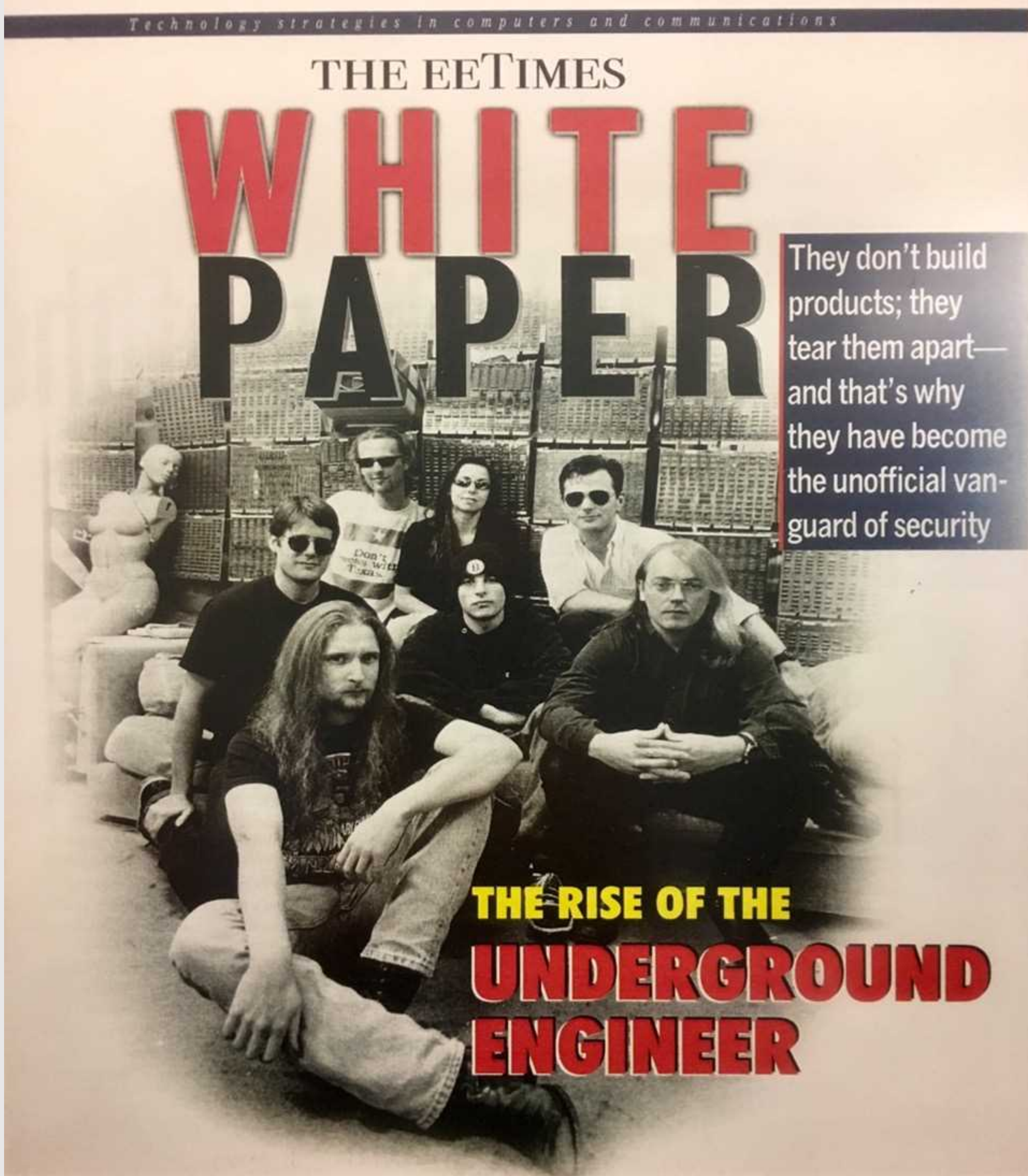
Into the light: *Once shadowy computer code warriors like Kingpin are going legit*

Using Good Hackers to Battle Bad Hackers

IF 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 "L0pht 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

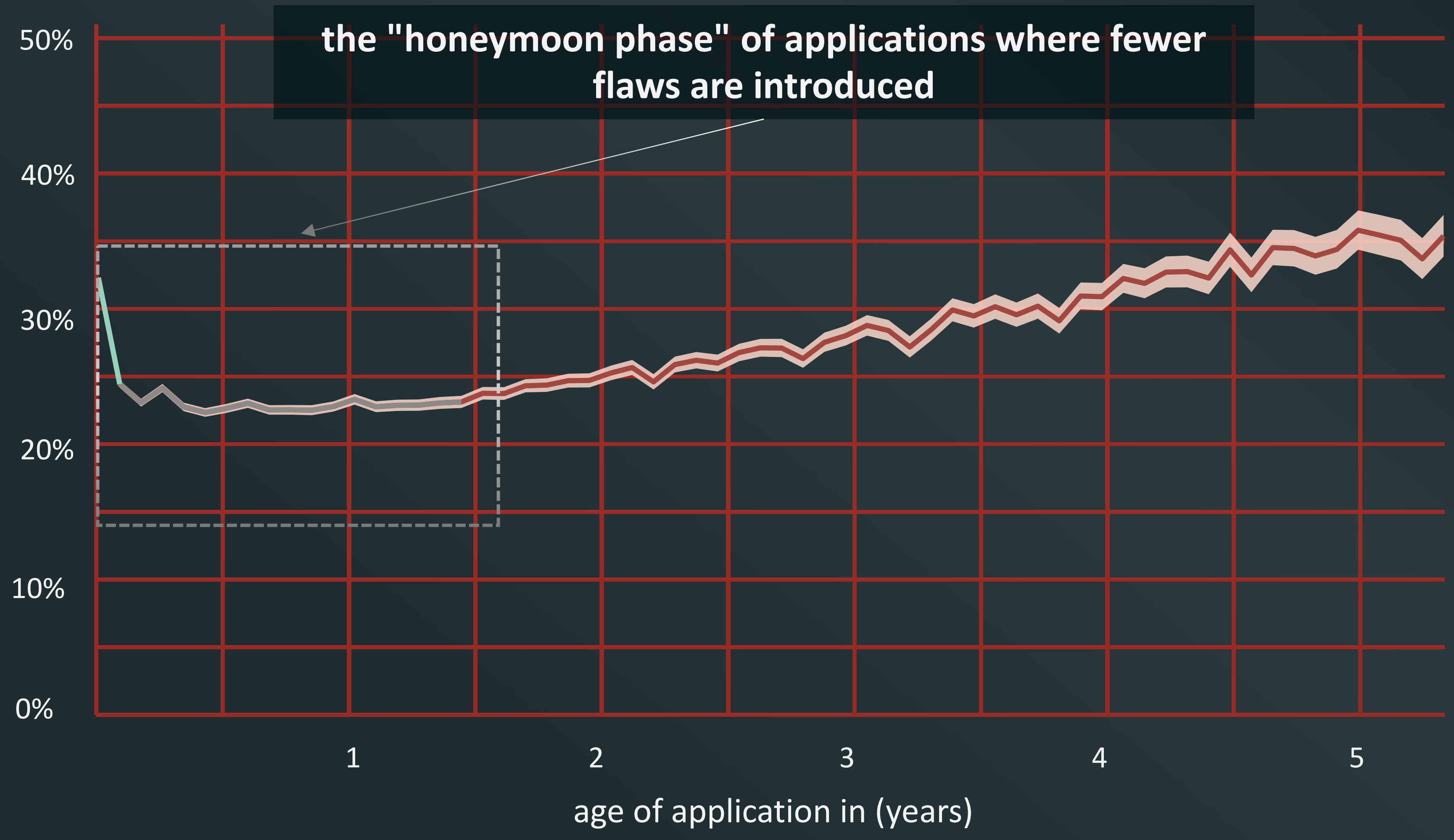


Founded @stake security research team and then Veracode to build security into SDLC

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

* We are defining all flaws that remain unremediated for over one year, regardless of severity, as security debt.
**Critical debt: High-severity flaws that remain unremediated for over one year.


2 out of 10
applications show an average monthly fix rate that exceeds ten percent of all security flaws.
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

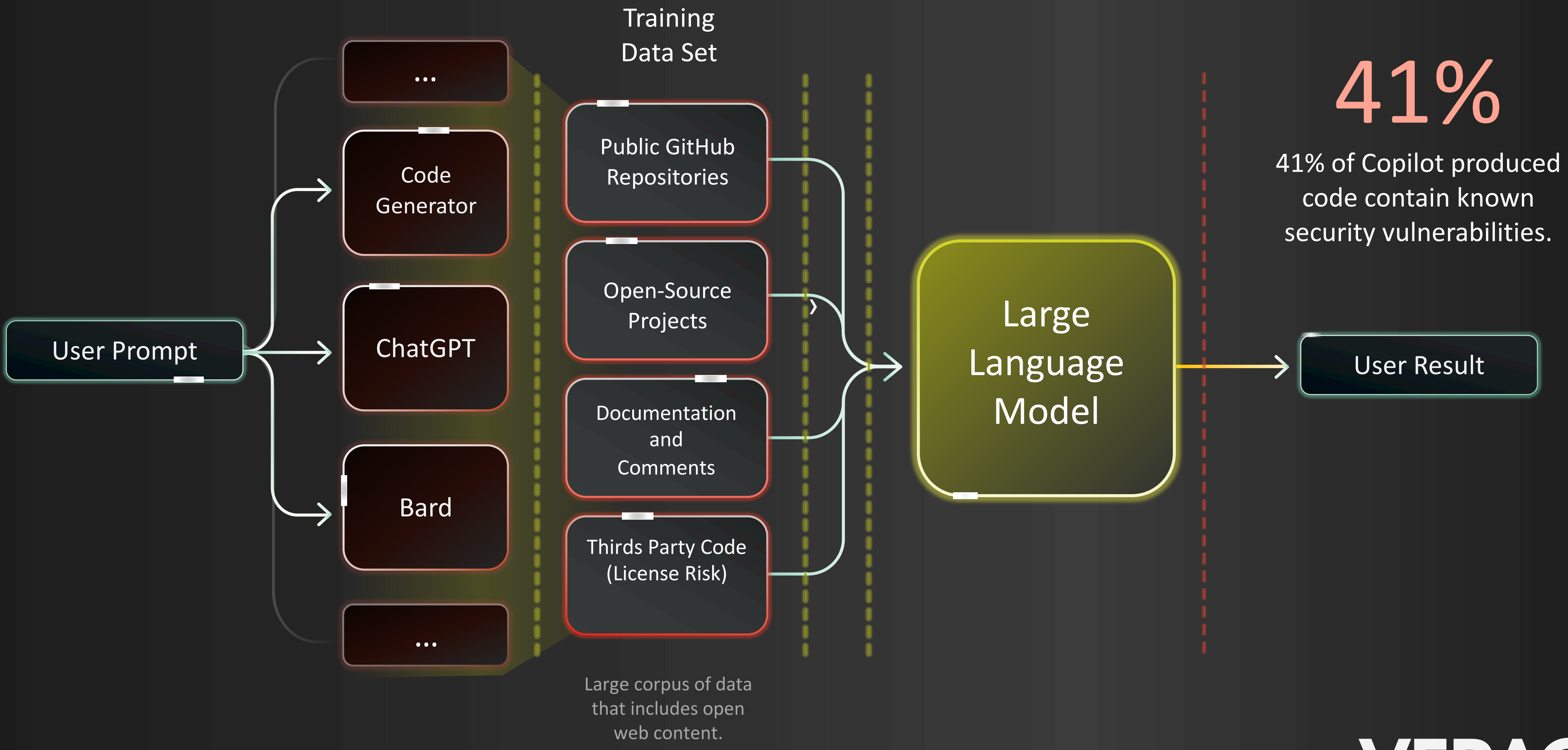
Searching for answers to avoid reinventing the wheel

Reading log files to find a root cause

Creating and running functional & non-functional tests

Remediating security vulnerabilities

Large Language Models



VERACODE

Security Implications of LLMs

Wuhan University Study
on AI Code Generators

36%

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

New York University Study
on GitHub Copilot

41%

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

Stanford University Study
on AI Code Generators

Developers using LLMs were more likely to write insecure code.

They were more confident their code was secure.

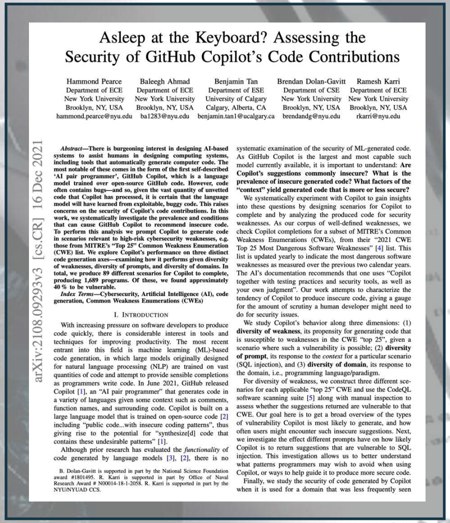
Purdue University
on ChatGPT accuracy

52%

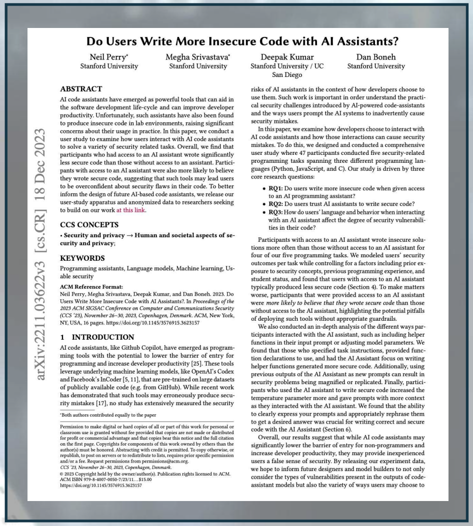
52% of ChatGPTs answers were incorrect.
Developers preferred them 35% of the time yet 77% of those answers were wrong


Security Weaknesses of Copilot Generated Code in GitHub

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


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

arXiv:2108.09293v3 [cs.CR] 16 Dec 2021


Do Users Write More Insecure Code with AI Assistants?

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


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

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

SALLM Framework For measuring LLM vulnerability generation - Notre Dame

Vulnerable@k metric best to worst:

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

VULNERABILITIES FOUND IN THE CHATGPT-GENERATED PYTHON CODES

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

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

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

***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.**

***Index Terms*—security evaluation, large language models, pre-trained transformer model, metrics**

I. INTRODUCTION

A *code LLM* is a Large Language Model (LLM) that has been trained on a large dataset consisting of both *text* and *code* [1]. As a result, code LLMs can generate code written in a specific programming language from a given *prompt*. These prompts provide a high-level specification of a developer's intent [2] and can include single/multi-line code comments, code expressions (*e.g.*, a function definition), text, or a combination of these. *etc.* Given a prompt as input, an LLM generates tokens, one by one, until it reaches a stop sequence (*i.e.*, a pre-configured sequence of tokens) or the maximum number of tokens is reached.

LLM-based source code generation tools are increasingly being used by developers in order to reduce software development efforts [3]. A recent survey with 500 US-based developers who work for large-sized companies showed that **92%** of them are using LLMs to generate code for work and personal use [4]. Part of this fast widespread adoption is due to the increased productivity perceived by developers; LLMs help them to automate repetitive tasks so that they can focus on higher-level challenging tasks [3].

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 **S**ecurity **A**ssessement of **LLM**s (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 (security@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[1];

[1]The dataset will be made public on GitHub upon acceptance.

https://arxiv.org/abs/2311.00889

Implications of LLM code generation

Code reuse goes down

Code velocity goes up

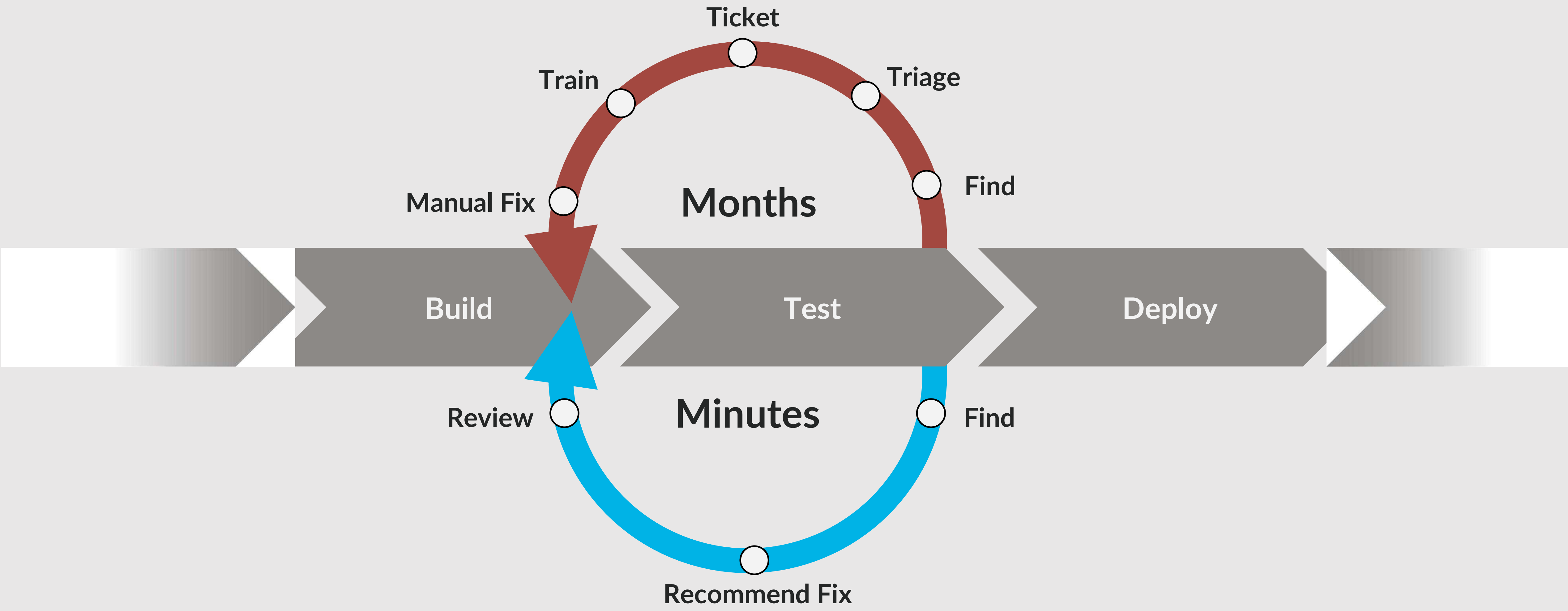
Vulnerability density similar

=

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)

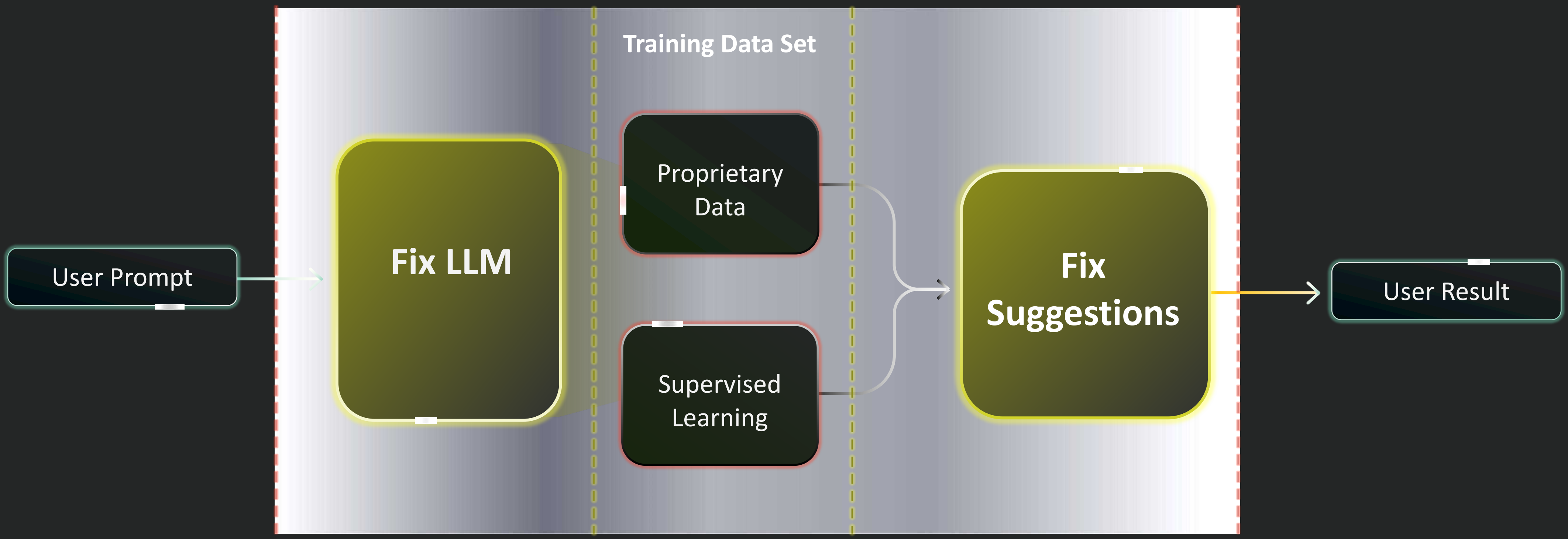
```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String name = req.getParameter("name");
    String[] array = new String[10];
    array[0] = name;
    PrintWriter writer = resp.getWriter();
    writer.println("Hello " + StringEscapeUtils.escapeHtml4(array[0]));
}
```

Fix Approach

Curated Dataset

Code Provenance Assurance

Coverage all that matter



VERACODE

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

Other Risks to GenAI Code

- Data Poisoning
- IP Infringement
- Recursive Learning
- Bias & Fairness
- Propagation of Deprecated Practices
- Hallucinated & Squatted Packages

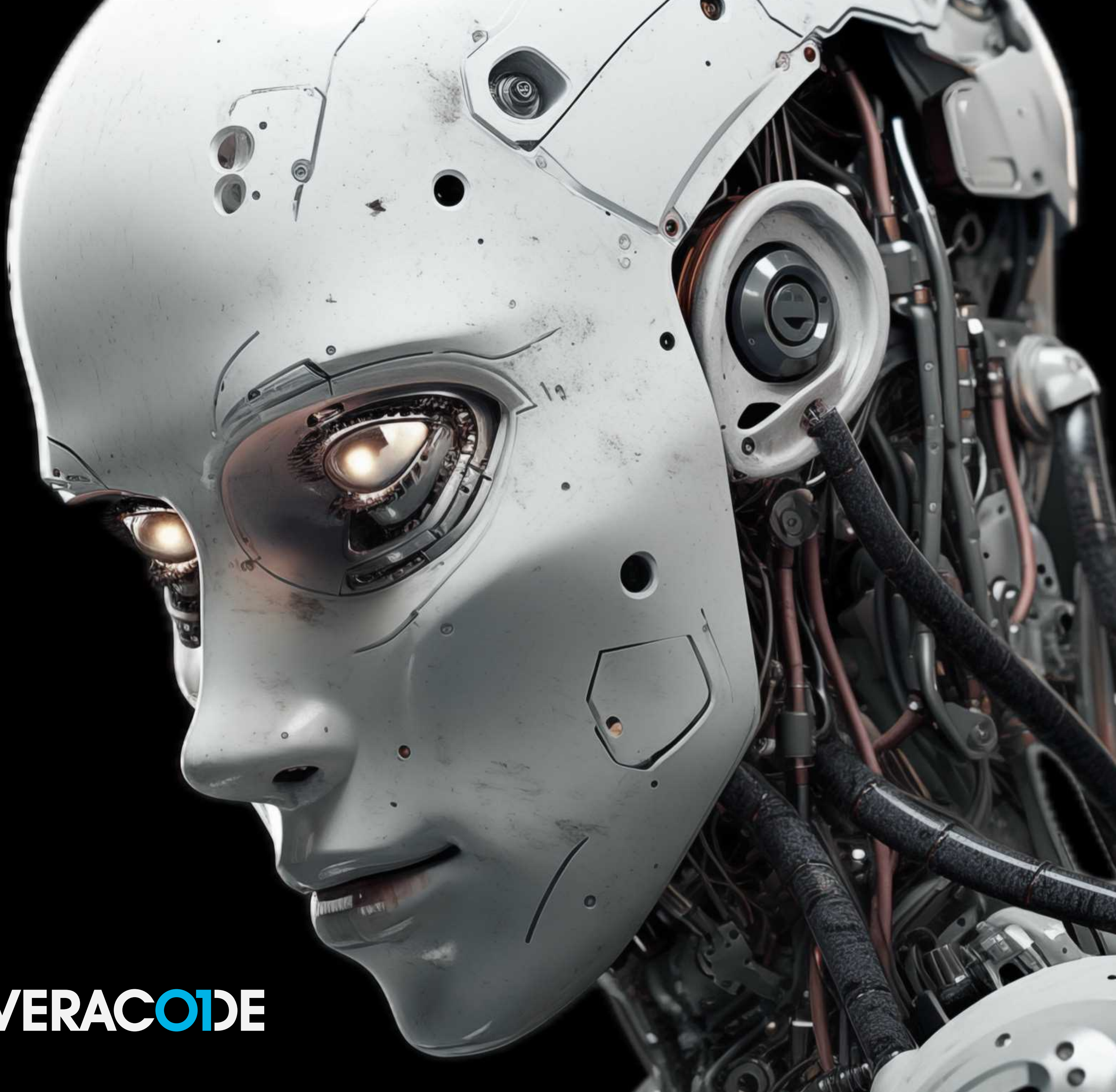
VERACODE

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