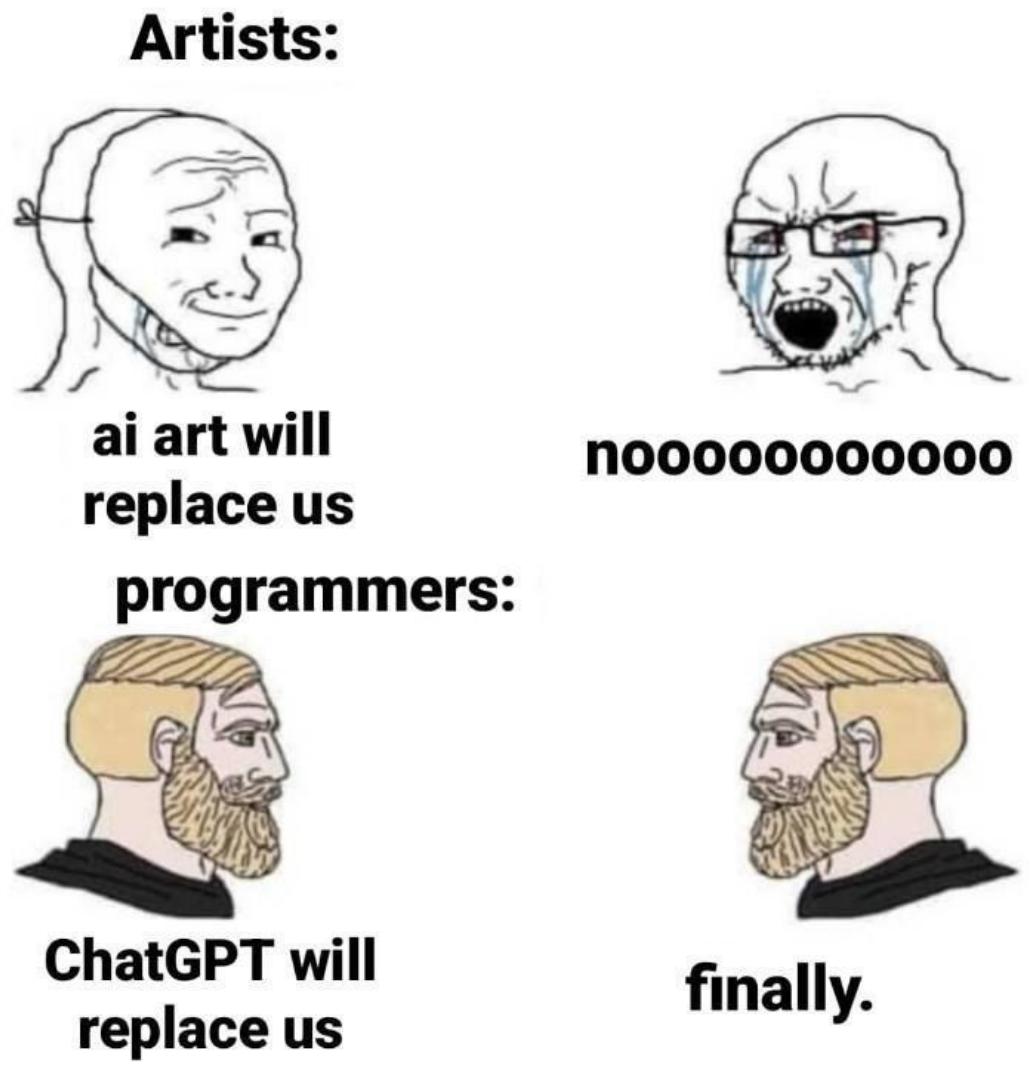
OPEN SOURCE LLMS FOR THE WIN

Asankhaya Sharma, CTO, Securade.ai

Rohan Sood, COO, Scantist

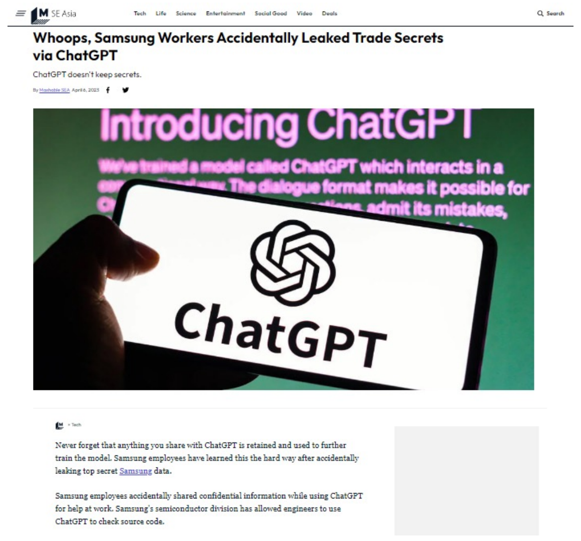
LARGE LANGUAGE MODELS FOR CODE

- CodeBERT
- AlphaCode
- CodeGen
- Codex
- GPT-3.5/4
- PanGu-Coder
- FauxPilot



WHY OPEN-SOURCE MODELS?

- Data security, privacy and control for organizations.
- Customization and optimization via fine-tuning/context embedding.
- Ability to train on a larger, ethically-sourced dataset.
- Greater transparency, avoids censorship and better alignment.
- Decentralization and reducing power concentration in AI development.
- Cost advantages and viability for non-commercial use-cases.



Whoops, Samsung Workers Accidentally Leaked Trade Secrets via ChatGPT

ChatGPT doesn't keep secrets.

By Mashable SEA April 6, 2023

Never forget that anything you share with ChatGPT is retained and used to further train the model. Samsung employees have learned this the hard way after accidentally leaking top secret Samsung data.

Samsung employees accidentally shared confidential information while using ChatGPT for help at work. Samsung's semiconductor division has allowed engineers to use ChatGPT to check source code.

A WINDOW OF OPPORTUNITY

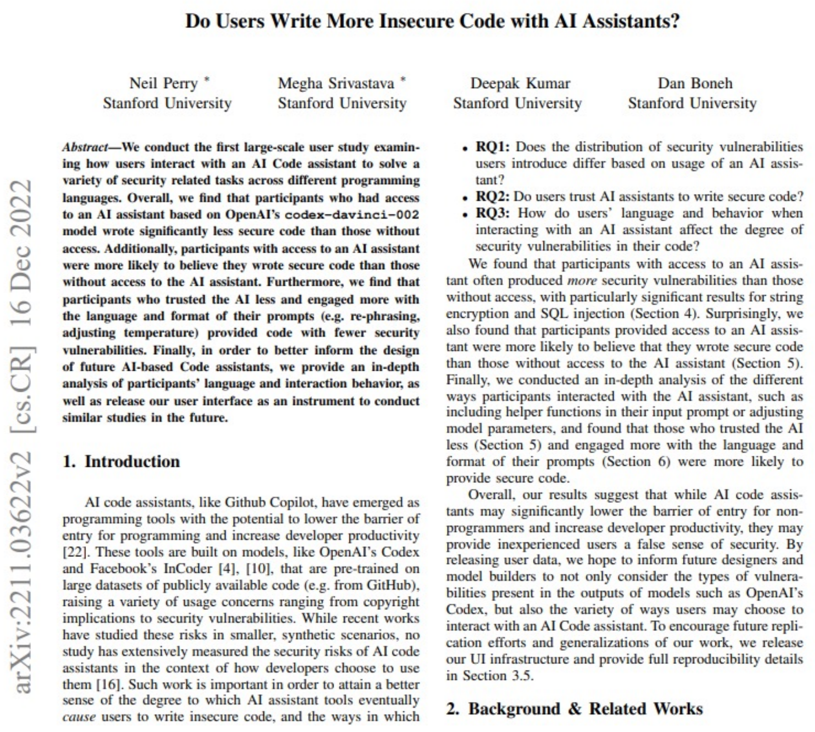
- Commercial models are leading when it comes to left-right code generation

Model	HumanEval pass @ 1
GPT-4	0.68
Codex	0.47
CodeGen	*0.29*
SantaCoder	*0.18*

- But open-source models are very competitive for fill-in-the-middle (FIM)

Model	FIM pass @ 1
Codex	0.63
PLBART	0.42
InCoder	0.49
SantaCoder	0.62

A WINDOW OF OPPORTUNITY



Do Users Write More Insecure Code with AI Assistants?

Neil Perry * Stanford University
Megha Srivastava * Stanford University
Deepak Kumar Stanford University
Dan Boneh Stanford University

***Abstract*—We conduct the first large-scale user study examining how users interact with an AI Code assistant to solve a variety of security related tasks across different programming languages. Overall, we find that participants who had access to an AI assistant based on OpenAI's `codex-davinci-002` model wrote significantly less secure code than those without access. Additionally, participants with access to an AI assistant were more likely to believe they wrote secure code than those without access to the AI assistant. Furthermore, we find that participants who trusted the AI less and engaged more with the language and format of their prompts (e.g. re-phrasing, adjusting temperature) provided code with fewer security vulnerabilities. Finally, in order to better inform the design of future AI-based Code assistants, we provide an in-depth analysis of participants' language and interaction behavior, as well as release our user interface as an instrument to conduct similar studies in the future.**

1. Introduction

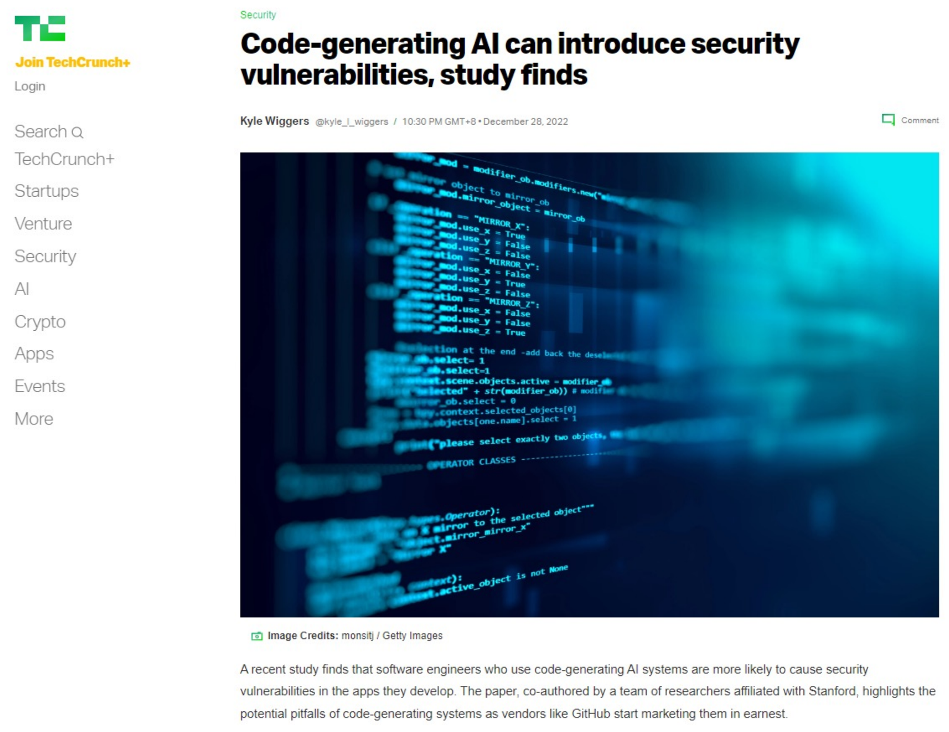
AI code assistants, like Github Copilot, have emerged as programming tools with the potential to lower the barrier of entry for programming and increase developer productivity [22]. These tools are built on models, like OpenAI's Codex and Facebook's InCoder [4], [10], that are pre-trained on large datasets of publicly available code (e.g. from GitHub), raising a variety of usage concerns ranging from copyright implications to security vulnerabilities. While recent works have studied these risks in smaller, synthetic scenarios, no study has extensively measured the security risks of AI code assistants in the context of how developers choose to use them [16]. Such work is important in order to attain a better sense of the degree to which AI assistant tools eventually *cause* users to write insecure code, and the ways in which

- **RQ1:** Does the distribution of security vulnerabilities users introduce differ based on usage of an AI assistant?
- **RQ2:** Do users trust AI assistants to write secure code?
- **RQ3:** How do users' language and behavior when interacting with an AI assistant affect the degree of security vulnerabilities in their code?

We found that participants with access to an AI assistant often produced *more* security vulnerabilities than those without access, with particularly significant results for string encryption and SQL injection (Section 4). Surprisingly, we also found that participants provided access to an AI assistant were more likely to believe that they wrote secure code than those without access to the AI assistant (Section 5). Finally, we conducted an in-depth analysis of the different ways participants interacted with the AI assistant, such as including helper functions in their input prompt or adjusting model parameters, and found that those who trusted the AI less (Section 5) and engaged more with the language and format of their prompts (Section 6) were more likely to provide secure code.

Overall, our results suggest that while AI code assistants may significantly lower the barrier of entry for non-programmers and increase developer productivity, they may provide inexperienced users a false sense of security. By releasing user data, we hope to inform future designers and model builders to not only consider the types of vulnerabilities present in the outputs of models such as OpenAI's Codex, but also the variety of ways users may choose to interact with an AI Code assistant. To encourage future replication efforts and generalizations of our work, we release our UI infrastructure and provide full reproducibility details in Section 3.5.

2. Background & Related Works



Security

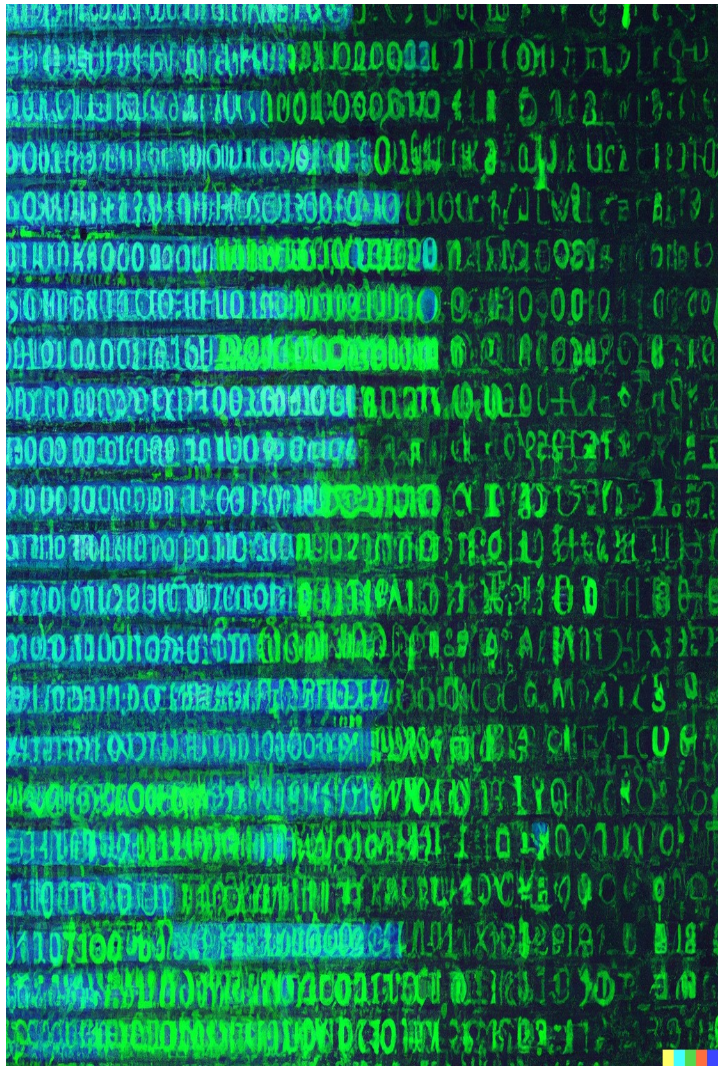
Code-generating AI can introduce security vulnerabilities, study finds

Kyle Wiggers @kyle_l_wiggers / 10:30 PM GMT+8 • December 28, 2022

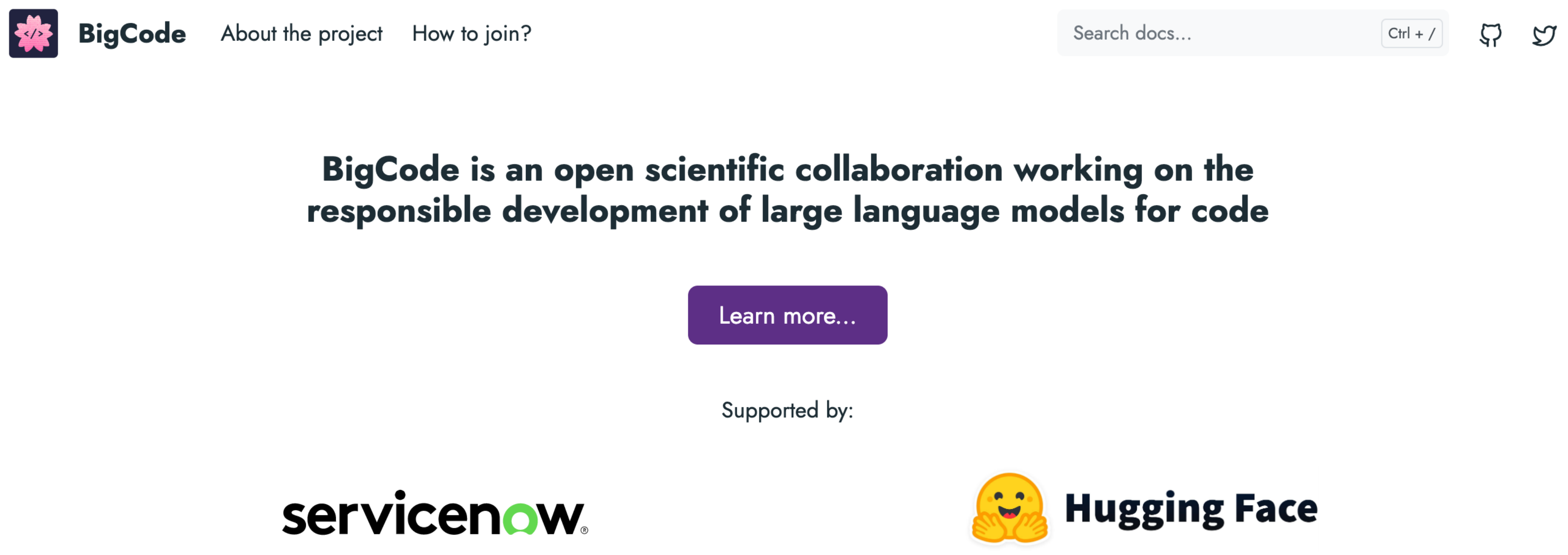
Image Credits: monsitj / Getty Images

A recent study finds that software engineers who use code-generating AI systems are more likely to cause security vulnerabilities in the apps they develop. The paper, co-authored by a team of researchers affiliated with Stanford, highlights the potential pitfalls of code-generating systems as vendors like GitHub start marketing them in earnest.

CAN OPEN-SOURCE LLMS FOR CODE FIX VULNERABILITIES?



SANTACODER



BigCode is an open scientific collaboration working on the responsible development of large language models for code

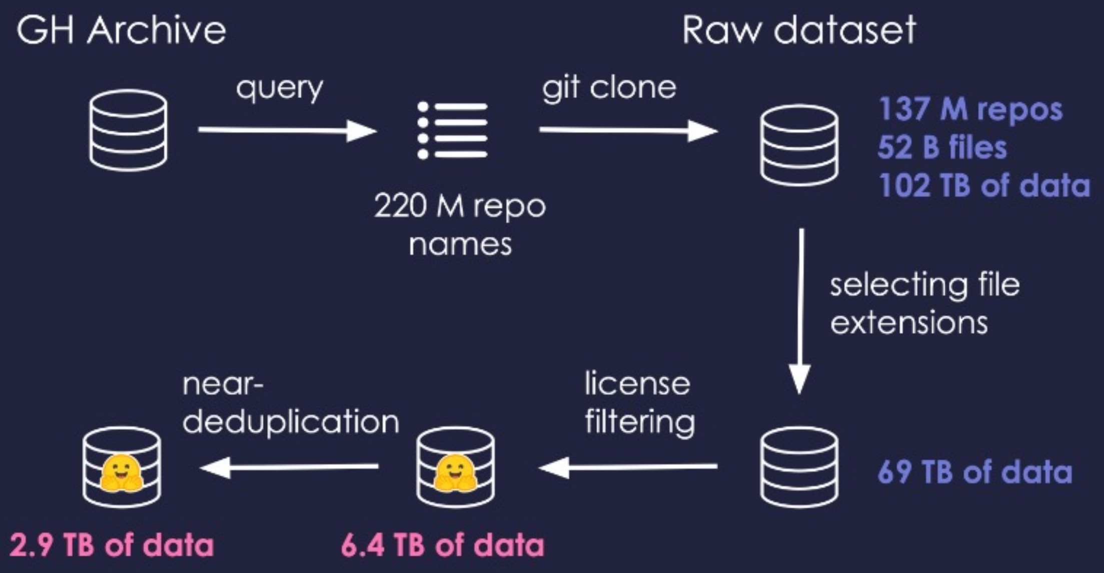
Supported by:

🎅 SantaCoder: don't reach for the stars! 🌟

The Stack

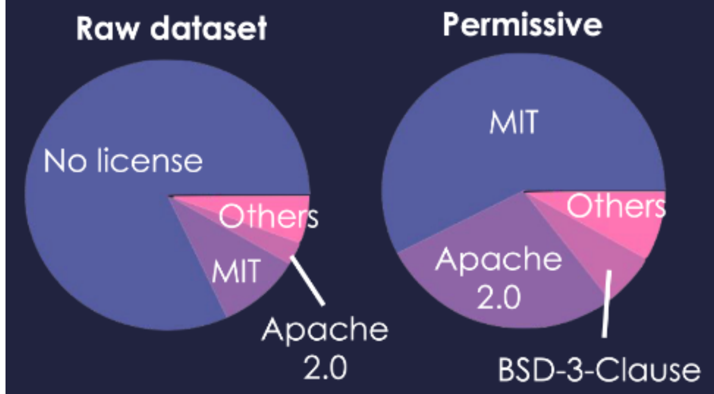
6 TB of permissive code data

Dataset Collection



Find the filtered and deduplicated datasets at: *www.hf.co/bigcode*

Licensing + Governance



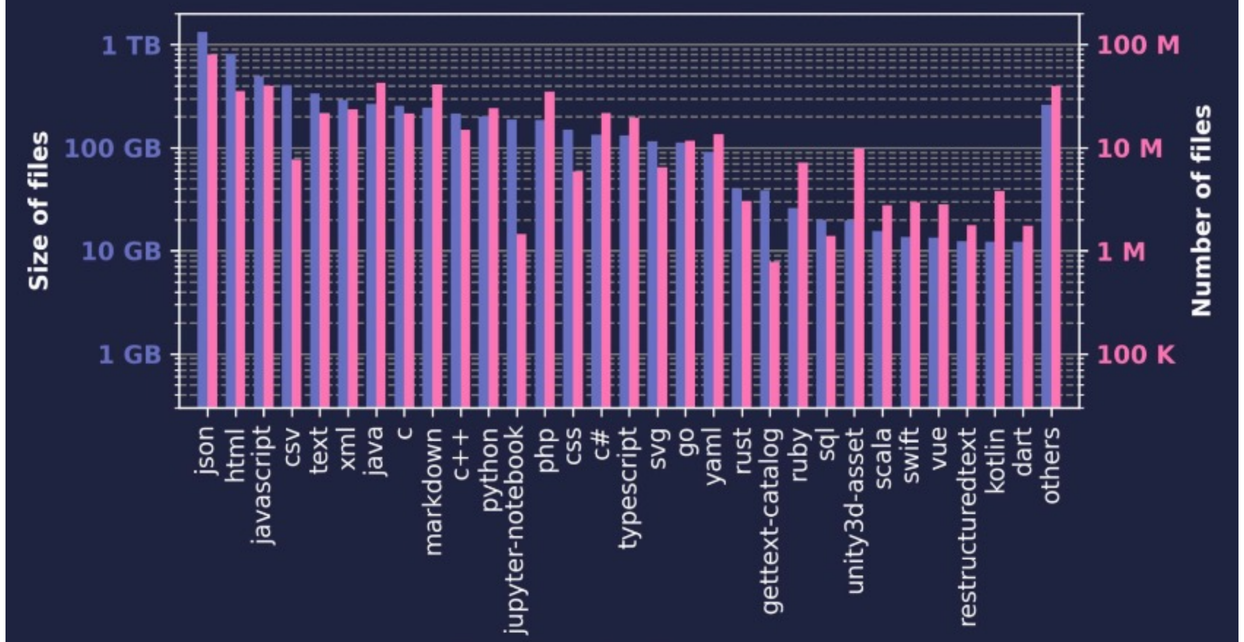
Opt-out: If users would like to exclude their code from the corpus we have an opt-out mechanism. Visit:

https://www.bigcode-project.org/docs/about/the-stack/

Permissive license distribution of licenses used to filter the dataset:

MIT (67.7%) | Apache-2.0 (19.1%) | BSD-3-Clause (3.9%) | Unlicense (2.0%) | CC0-1.0 (1.5%) | BSD-2-Clause (1.2%) | CC-BY-4.0 (1.1%) | CC-BY-3.0 (0.7%) | 0BSD (0.4%) | RSA-MD (0.3%) | WTFPL (0.2%) | MIT-0 (0.2%) | Others (166) (2.2%)

Programming Languages



Evaluation

We trained several **GPT-2 models (350M parameters)** on different parts of the dataset both with and without near-deduplication. The models trained on the Python subset of The Stack performed on par with CodeX and CodeGen of similar size when using near-deduplication.

Dataset	Filtering	pass@1	pass@10	pass@100
Codex (300M)	unknown	13.17	20.17	36.27
CodeGen (350M)	unknown	12.76	23.11	35.19
Python all-license	None	13.11	21.77	36.67
	Near-dedup	**17.34**	**27.64**	**45.52**
Python permissive-license	None	10.99	15.94	27.21
	Near-dedup	**12.89**	**22.26**	**36.01**

**results obtained with The Stack v1.0*

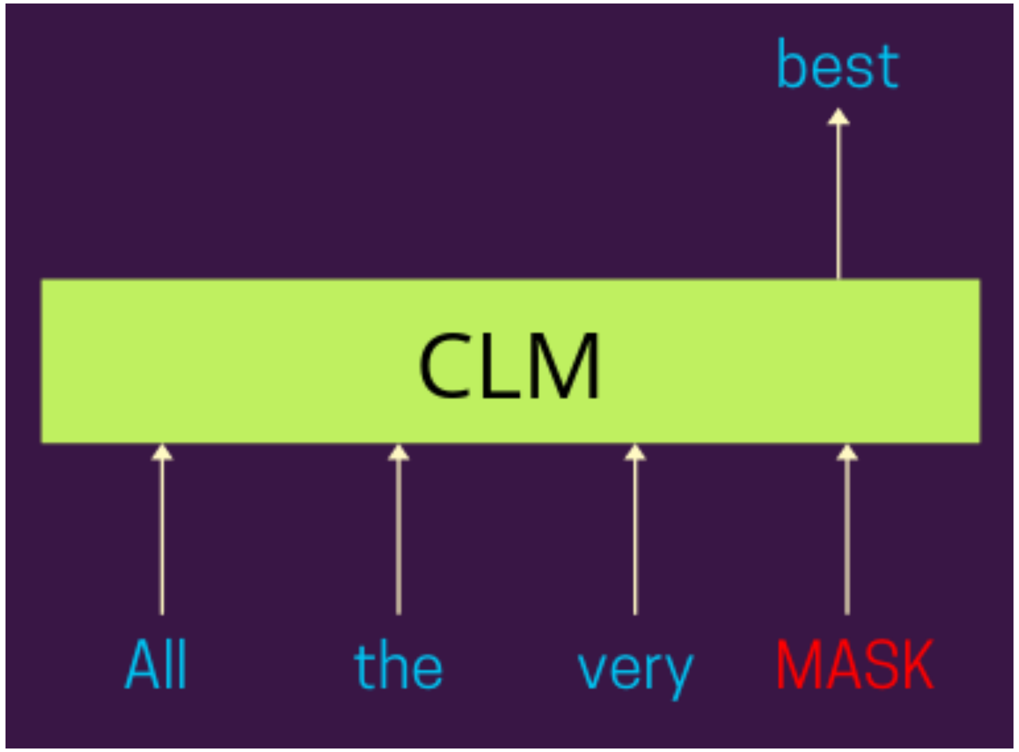
@BigCodeProject

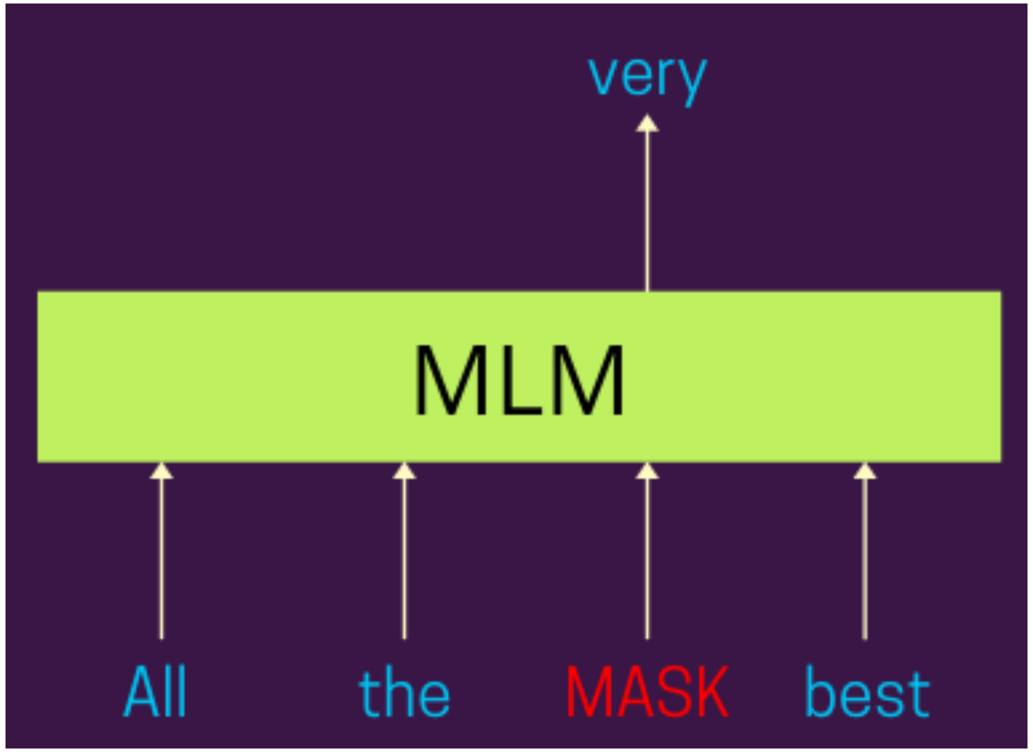
https://www.bigcode-project.org/

contact@bigcode-project.org

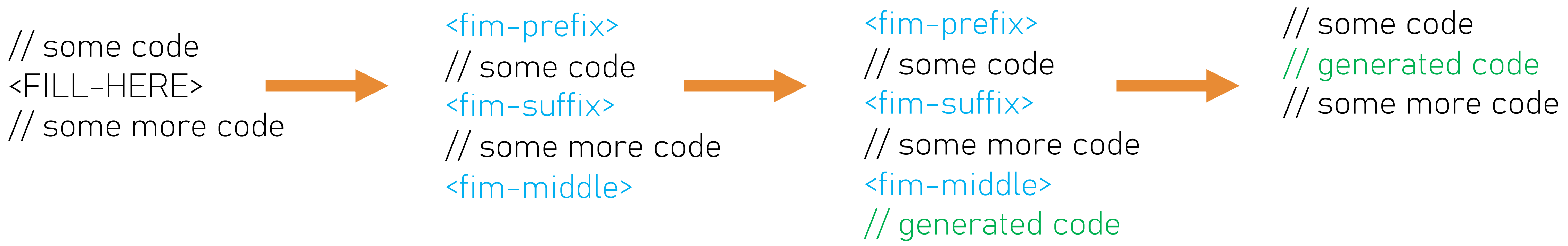
SUPERVISED FINE TUNING

- GPT2 fine tuning for CLM (causal language modeling)





INFILLING AT INFERENCE WITH CLM



```
// some code
<FILL-HERE>
// some more code
```

```
<fim-prefix>
// some code
<fim-suffix>
// some more code
<fim-middle>
```

```
<fim-prefix>
// some code
<fim-suffix>
// some more code
<fim-middle>
// generated code
```

```
// some code
// generated code
// some more code
```

BUG FIXING

- Prepare the data for fine-tuning

```
// some code
// buggy line
// some more code
```

```
// some code
// BUG: CWE ID: DESC
// buggy line
// FIXED:
// fixed line
// some more code
```

```
<fim-prefix>
// some code
// BUG: CWE ID: DESC
// buggy line
// FIXED:
<fim-middle>
// fixed line
<fim-suffix>
// some more code
<|endoftext|>
```

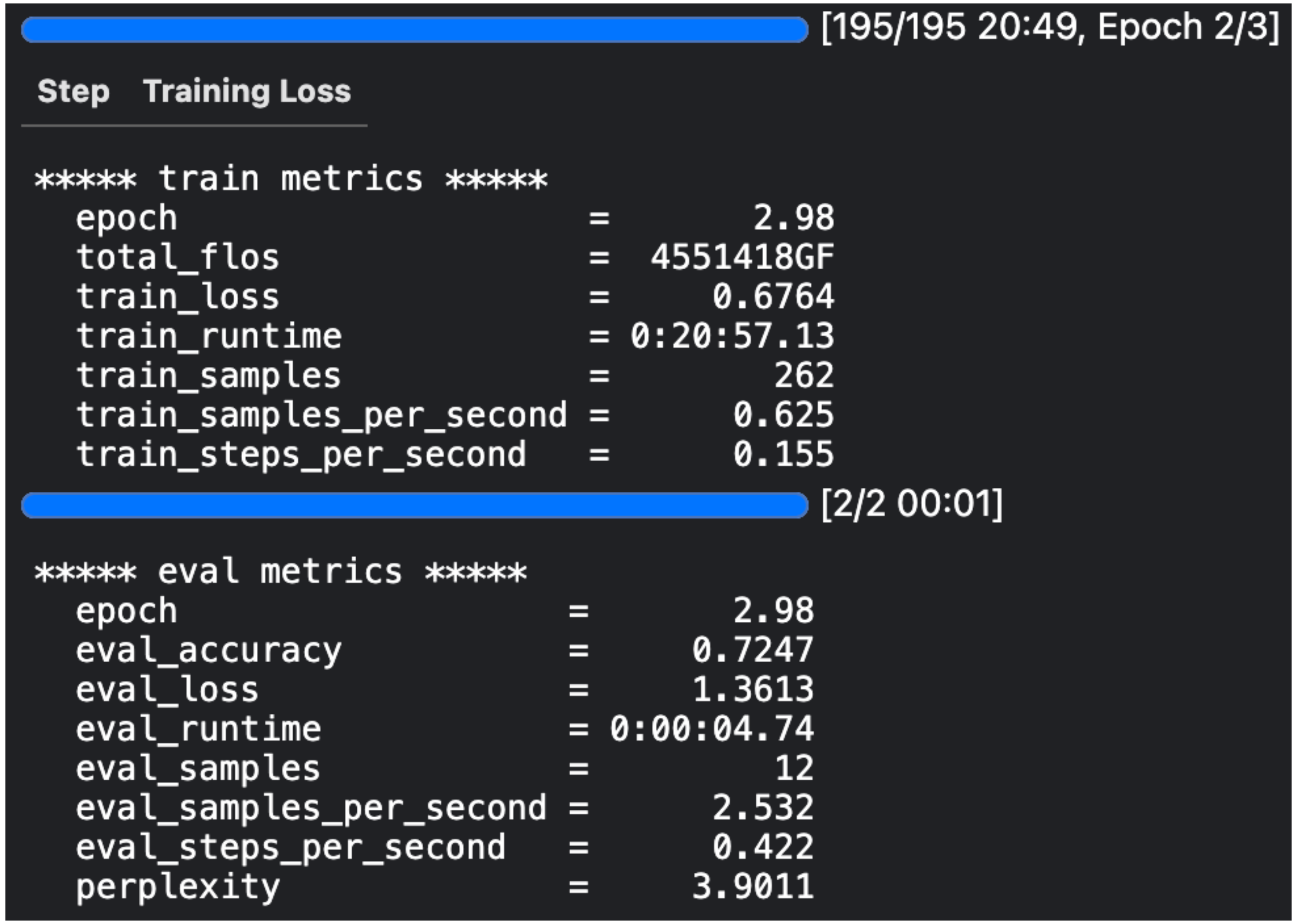
EXAMPLE

```
String output = Launcher.RESOURCES.getString("WinstoneResponse.ErrorPage",
// BUG: CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
// new String[] { sc + "", (msg == null ? "" : msg), sw.toString(),
// FIXED:
new String[] { sc + "", URIUtil.htmlEscape(msg == null ? "" : msg),

URIUtil.htmlEscape(sw.toString()),Launcher.RESOURCES.getString("ServerVersion"),"" + new Date() });

response.setContentLength(output.getBytes(response.getCharacterEncoding()).length);
Writer out = response.getWriter();
```

TRAINING

- Add special tokens

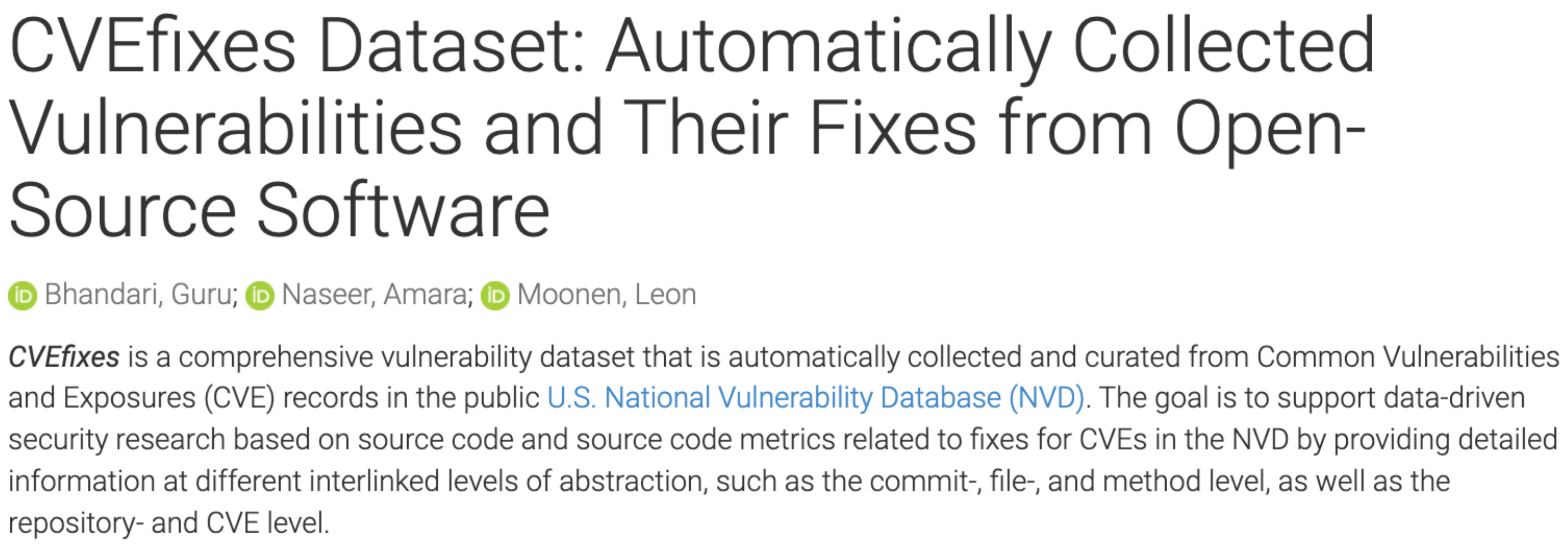
```
FIM_PREFIX = "<fim-prefix>"
FIM_MIDDLE = "<fim-middle>"
FIM_SUFFIX = "<fim-suffix>"
FIM_PAD = "<fim-pad>"
EOD = "<|endoftext|>"
tokenizer.add_special_tokens({
    "additional_special_tokens": [EOD,
FIM_PREFIX, FIM_MIDDLE, FIM_SUFFIX,
FIM_PAD],
    "pad_token": EOD,})
```



```
[195/195 20:49, Epoch 2/3]
Step  Training Loss

***** train metrics *****
  epoch                    =        2.98
  total_flos               =    4551418GF
  train_loss               =      0.6764
  train_runtime            = 0:20:57.13
  train_samples            =         262
  train_samples_per_second =       0.625
  train_steps_per_second   =       0.155
[2/2 00:01]

***** eval metrics *****
  epoch                   =        2.98
  eval_accuracy           =      0.7247
  eval_loss               =      1.3613
  eval_runtime            = 0:00:04.74
  eval_samples            =          12
  eval_samples_per_second =       2.532
  eval_steps_per_second   =       0.422
  perplexity              =      3.9011
```

SANTAFIXER

- Trained on a dataset of single line CVE fixes
- Dataset is available at https://huggingface.co/datasets/lambdasec/cve-single-line-fixes



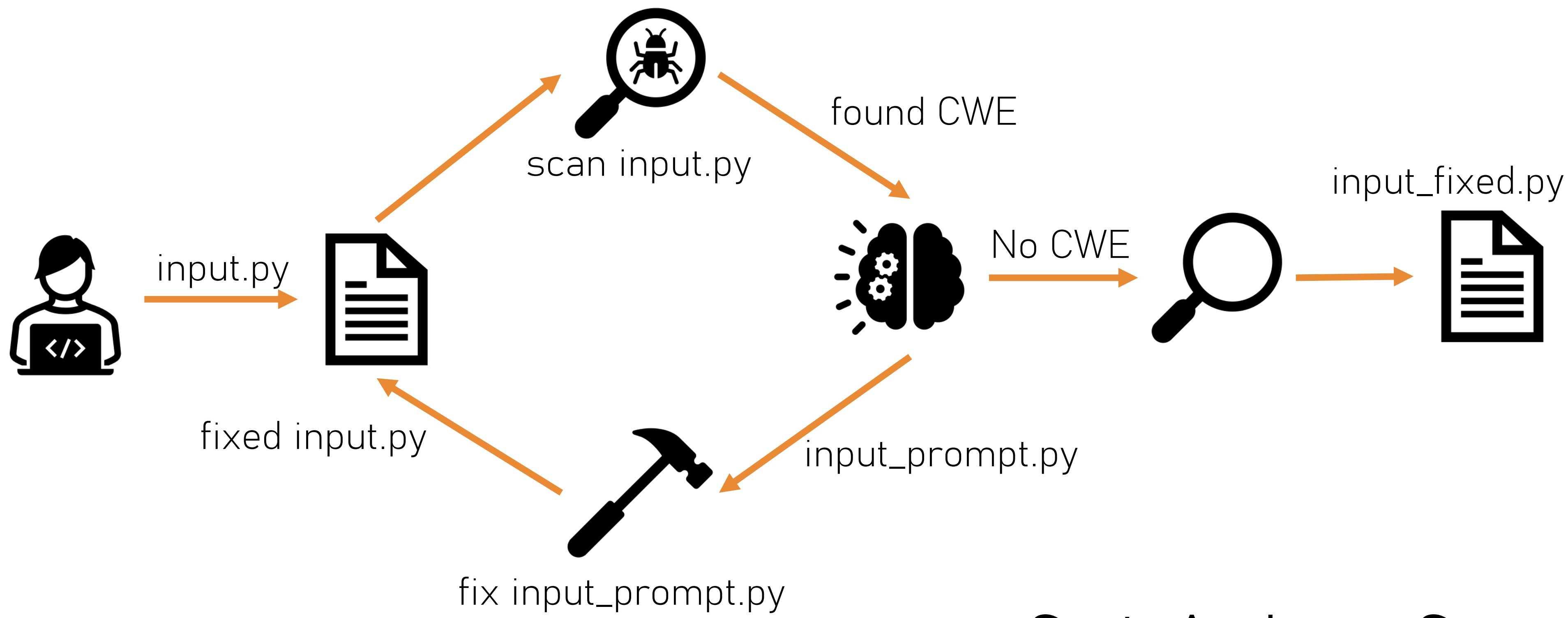
CVEfixes Dataset: Automatically Collected Vulnerabilities and Their Fixes from Open-Source Software

Bhandari, Guru; Naseer, Amara; Moonen, Leon

CVEfixes is a comprehensive vulnerability dataset that is automatically collected and curated from Common Vulnerabilities and Exposures (CVE) records in the public U.S. National Vulnerability Database (NVD). The goal is to support data-driven security research based on source code and source code metrics related to fixes for CVEs in the NVD by providing detailed information at different interlinked levels of abstraction, such as the commit-, file-, and method level, as well as the repository- and CVE level.

SantaFixer : https://huggingface.co/lambdasec/santafixer

STATIC ANALYZER + LLM = AUTO FIX



Static Analyzer : Semgrep
LLM : SantaFixer

AutoFix : https://github.com/lambdasec/autofix

RESULTS

- Evaluated on a dataset of CVEs from top 1000 projects on GitHub, scanned using Semgrep
- Dataset is available at https://huggingface.co/datasets/lambdasec/gh-top-1000-projects-vulns

GH Top 1000 Projects Vulns	pass @ 1	pass @ 10
Java	0.26	0.48
Python	0.31	0.56
JavaScript	0.36	0.62

THANKS

- Questions?
- Hugging Face
  - https://huggingface.co/lambdasec
- GitHub
  - https://github.com/lambdasec/autofix