W&M ScholarWorks

Undergraduate Honors Theses

Theses, Dissertations, & Master Projects

5-2024

Security and Interpretability in Large Language Models

Lydia Danas
William & Mary

Follow this and additional works at: https://scholarworks.wm.edu/honorstheses

Part of the Artificial Intelligence and Robotics Commons, Data Science Commons, and the Information Security Commons

Recommended Citation
Danas, Lydia, "Security and Interpretability in Large Language Models" (2024). *Undergraduate Honors Theses.* William & Mary. Paper 2227.
https://scholarworks.wm.edu/honorstheses/2227

This Honors Thesis -- Open Access is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Security and Interpretability in Large Language Models

A thesis presented in Candidacy for Departmental Honors in
Computer Science
from
the College of William and Mary in Virginia

By
Lydia M. Danas
May 6th, 2024

Accepted for Honors

[illegible]
Prof. Denys Poshyvanyk

Adwait
Prof. Adwait Nadkarni

[illegible]
Prof. Martin White

Cristiano Fanelli
Prof. Cristiano Fanelli

Security and Interpretability in Large Language Models

Lydia M. Danas

May 2024

Contents

Acknowledgments

I would first like to extend most sincere thanks and gratitude to Dr. Poshyvanyk for being my advisor, and to Dr. Fanelli, Dr.White, and Dr. Nadkarni for serving on my examining committee. Your passion for and deep interest in the advancement of computer and/or data science is a true inspiration and I could not be more grateful for your feedback and guidance. It has been a true honor to work with and learn from you during my time as an undergraduate.

Next I would like to sincerely thank Alejandro and Daniel for always providing poignant questions and comments (especially relating to data)! I wish you the best of luck in your future studies and endeavors.

Third, I would like to extend many heartfelt thanks to my parents for their constant and unwavering support of my passions. I would also like to extend these thanks to my Aunt Mary Ann and Aunt Lauren, whose passions are always an inspiration to strive for excellence.

Lastly, I would like to extend many thanks to my friends, who have let me yap to them about this thesis ad nauseum. An extra special thanks to Georgia Carpenter for always swemming with me.

List of Figures

Abstract

Large Language Models (LLMs) have the capability to model long-term dependencies in sequences of tokens, and are consequently often utilized to generate text through language modeling. These capabilities are increasingly being used for code generation tasks; however, LLM-powered code generation tools such as GitHub's Copilot have been generating insecure code and thus pose a cybersecurity risk. To generate secure code we must first understand why LLMs are generating insecure code. This non-trivial task can be realized through interpretability methods, which investigate the hidden state of a neural network to explain model outputs. A new interpretability method is rationales, which obtains the minimum subset of input tokens that lead to the model's output. Through obtaining rationales of insecure code, we are able to investigate the relationship between model inputs and LLM-generated insecure code tokens to further efforts in mitigating cybersecurity risks currently posed by LLM-generated code.

Our experiment conducts a case study on two common, pervasive, and severe real-world weaknesses: XSS injection (CWE-79) and SQL injection (CWE-89). We first collected data, then obtained rationales for our weak Python code samples via the greedy rationalization algorithm and a GPT-2 model. Thus, we were able to identify the specific tokens which lead to insecure token generation. We also explored an aggregation function for code rationales - structural code taxonomy - which allowed us to investigate rationales on the local and global levels. Our prototype study found good results: rationales for CWE-79 and CWE-89 code samples have different structural code taxonomy mappings. This implies that each LLM-generated weakness arises from different aspects of the code context, and thus efforts to mitigate insecure LLM-generated code must be precisely targeted to the weakness.

Chapter 1

Introduction

You taught me language, and my profit on't
Is I know how to curse. The red plague rid you
For learning me your language!

The Tempest
William Shakespeare

1.1 The OpenAI Effect

On November 30th, 2022, OpenAI introduced the world to ChatGPT. By December 6th the large-language model powered chatbot - which was still a demo at the time - had garnered one million users [1]. Machine learning is nothing new, but with ChatGPT's immense popularity came increased awareness and application of machine learning to pretty much everything: art, predictive analytics, admissions departments, social media, email drafting, the classroom, and software engineering.

OpenAI launched ChatGPT with an example of a developer asking the chatbot for help with debugging a block of code [2]. The idea of teaching a generative large language model to code has been a pursuit of computer scientists and researchers for years. This area of research treats programming languages as what they are - languages. In this context, we can treat code as sequences of tokens with varying meanings and apply language modeling techniques to generate code based on an input sequence. Such input sequences are known as prompts or contexts, and the deep models learning the long- and short-range relationships between tokens are known as neural language models [3]. Have the recent quantum leaps in the field of neural language modeling enabled the practical realization of a code generation model? Perchance.

1.2 Where Things Can Go Wrong

A few weeks ago, a friend of mine overheard this insult outside of Swem: "She was completely AI generated". Evidently if GenAI is being used as an insult it is not perfect. Some pitfalls are outputs that lack necessary nuance, detail, or are simply downright incorrect [4]. Code generation tasks are not exempt from these problems. Although the generation of syntactically correct code has improved in recent years as evidenced by the rising popularity of code generation tools [5, 6, 7, 8], other concerns have risen; most critical of these is the security of large language models [9]. Generated code can behave in such a manner that the security of the code is flawed - in other words, generated code can contain insecurities [10]. Such insecure code when implemented in a product might be exploitable by end users, which opens the door to any number of cybercrimes potentially resulting in compromised filesystems and personally identifiable information, hacked Facebook accounts, or malware installation [11]. Unfortunately, teaching a model to generate secure code is not an easy feat; as such, preventing insecure code generation necessitates understanding the factors that can lead to it.

Enter cybersecurity, which helps us categorize, understand, and fix insecure code; interpretability, which helps us understand how a model arrives at a prediction; and rationales, a new interpretability method of analyzing what input tokens generate the aformentioned prediction. Through combining these three concepts, we can investigate what aspects of the learned relationship between tokens results in insecure code generation.

Our study investigates insecure code generation by LLMs through application of a new technique called sequential rationales [12] in a case study on two types of insecure Python code. This interpretability technique extracts the smallest subset of tokens in model prompts which produce the output - when applied to insecure tokens, we can create rationales for insecure code. By aggregating and mapping these tokens to an AST-based structural code taxonomy, we are able to identify what types of tokens most influence generation of CWE-79 (XSS injection) and CWE-89 (SQL injection). We apply this technique through the greedy rationalization algorithm after compatibalizing the GPT-2 based Codeparrot-small model.

1.3 Overview

In the next chapter we will discuss related work. Chapters 3-5 provide the necessary background for our study. In Chapter 3, we discuss cybersecurity with a particular focus on the two Common Weakness Enumerations (CWEs) used for our case study. Chapter 4 delves into language modeling and its particular use for code generation tasks, and Chapter 5 provides the theory underlying interpretability methods in conjunction with an explanation of rationales. After this, we discuss details of our sequential rationales implementation with the context of interpretability in Chapter 6, followed by an exhaustive introduction of our data in Chapter 7. We then reach our results in Chapter 8 which contains global and local analysis of rationales and potential implications for secure code generation efforts. Finally, Chapter 9 is our conclusion. We also include two appendices, which contain additional exploratory data analysis on our source datasets and general information on state of the art LLM architectures.

Chapter 2

Related Work

Language Modeling in Software Engineering Natural language refers to languages which develop in an unplanned manner from a community or group of people [13]. Statistical processes are able to model natural languages with large, nuanced vocabularies because their use is typically repetitive and thus predictable. In 2012, Hindle et al. proposed that software can be treated as a natural language for code generation because software is also used in a repetitive and predictable matter [14]. As the machine learning field became more popular over the following years, more research explored the capabilities of NLMs [3]; in 2015, White et al. explored the application of NLMs to code, outperforming previous n-gram models [15]. From here, research interests have diverged. One focus is use in software engineering tasks such as code completion and translation [3]. Another focus is exploring possible representations of LMs for code [16, 17]. Regardless of the specific LM representation, it is clear that SOTA performance across multiple software engineering tasks is obtained by using a LM to represent code followed by targeted finetuning for the specific task [14, 18, 19]. This is often actualized through use of a large language model (LLM) [3].

Security of LLM-Generated Code However, LLM-generated code can contain insecurities - particularly when trained on real code which can contain bugs or exploitable weaknesses known as vulnerabilities. This can become a pervasive problem for the aforementioned increasingly popular LLM-based SE tools used for code completion and translation. Pearce et al. examine the conditions required for insecure code generation by one such SE tool, GitHub's AI assistant programmer Copilot. This examination revealed that almost half of the generated code created under conditions intended to potentially lead to insecurity contained a MITRE Top 25 CWE weakness [9]. Insecure code which impacts downstream tasks is extremely undesirable for SE purposes; thus, He et al. introduced a twofold approach to minimizing insecure code generation through controlled code generation. This new method leverages adversarial testing and security hardening to control the output of a model without altering the model parameters themselves [20].

Interpretability for LLMs Preventing insecure code generation first starts with understanding insecure code generation; to do this, we must understand why a given model generates its outputs. Interpretability is a collection of methods which make this possible through investigating the inner workings of the model (i.e., hidden state h_t), thereby explaining model predictions [21]. Much recent work has investigated the importance and impact of varying granular structural information inclusion in such methods for pre-trained NCMs. Palacio et al. determined NCMs heavily emphasize code syntax and as such changes in syntax influence model predictions; because code syntax is inherently structural, this finding underscores the importance of including structural information in interpretability methods [21]. Wan et al. concluded the same from a comprehensive study on two specific NCMs, CodeBERT and GraphCodeBERT. They explored the role of word embeddings, self-attention, and induction in the models, determining that self-attention weights were similar to syntactic structures, pre-training results in hidden state syntactic structure retention, and pre-trained NCMs can induce syntax trees [22]. Palacio et al. expand upon this last finding in 2023 through creation of a tool (ASTxplainer) which treats the syntax structures defined by ASTs as human-interpretable concepts. This tool allows for practical interpretation of NCM predictions through using cross-entropy loss to aggregate predicted tokens into their corresponding AST node type [23].

Explainability for LLMs Closely related to interpretability studies are explainability studies, which also aim to make sense of model predictions [23]. One common explainability method is self-attention, which is easily adaptable for interpretability purposes due to its inherent human-understandable nature [24]. However, the accuracy and validity of explanations provided by this method is hotly debated. Some experiments conclude the method is reliable [25] [26], while others argue it is not [27] [28]. Enter Bastings et al., who partially closed this discussion by arguing attention's easy application to explainability is merely a coincidence and that the variety of existing post-hoc input saliency methods are more useful [24]. Thus, using post-hoc interpretability techniques results in more comprehensive explanations of model behavior.

Chapter 3

Cybersecurity

Broadly speaking, cybersecurity is the shield that protects technology from malicious actors. These malicious actors commit cybercrime, which is made up of cyber attacks intended to compromise the target computer program, system, application, or hardware. The field of cybersecurity is defined by the prevention of and response to cyber attacks [29]. The vulnerability theory framework supplies the cybersecurity field with precise and substantive vocabulary for vulnerability discussions [11]. As such, it is relevant to our discussion of cybersecurity, which depends on the definition of many terms in vulnerability theory.

3.1 Vulnerability Theory

A **product** is a software package, protocol design, or architecture, which offers some main capability or feature. This product implements said features by performing behaviors which operate on resources. **Behaviors** are an action the product takes or user performs to provide the features, and **resources** are an entity that is used, modified or provided by the product (such as a CPU). Products may restrict the access of certain resources or behaviors to specific actors or group of actors; in this case, these **control spheres** are implicitly or explicitly defined by a **security policy**. When the product's intended security policy is not the same as the implemented security policy, a **vulnerability** could exist. **Actors** are an entity which interacts with the product or other entities who interact with it. These **entities** can be any combination of a product, person, or process. When an actor attempts to violate the intended security policy, the product has been **attacked** [11].

Any code that could lead to violation of its security policy is considered insecure. Such insecure code contains weaknesses. When these weaknesses are exploitable by an end user (actor), they are called vulnerabilities and the code permits the violation of its security policy. A vulnerability span is the initial and final row and column locations of the insecure tokens which constitute the vulnerability or weakness. Extracting the vulnerability span can be done with automated tools, but is most reliable when done by manual code review. Vulnerabilities can be "fixed" by any software change that either completely resolves the vulnerability or mitigates its impacts [10, 11, 30].

In sum: Code that could lead to security issues is insecure and can contain any number of weaknesses or vulnerabilities. When deployed, exploitable weaknesses are fixable vulnerabilities. These vulnerabilities can result in violated security policies, thus providing malicious actors access to certain behaviors or resources provided by the product [10, 11, 30]. Determining if code poses a cybersecurity risk is a complicated task. Three relevant organizations providing tools, data, and terminology to help with this are OWASP, NIST, and MITRE.

3.2 Relevant Organizations

OWASP The **O**pen **W**orldwide **A**pplication **S**ecurity **P**roject (OWASP) is a nonprofit foundation which aims to increase software security through a number of initiatives. These initiatives are designed to raise awareness about cybersecurity and mitigate cybersecurity risks, and can be anything from events to organizing chapters of volunteers and individual projects [31]. The most popular OWASP project is the OWASP Top 10, which raises awareness about critical web application-related security risks through a ranked list which is updated roughly every 3-4 years. This list is created with an emphasis on CWE mappings, which are then placed into OWASP-specific categories [32]. Another relevant project is the OWASP Top 10 for LLMs. Founded in May 2023, this project is an analog of the OWASP Top 10: instead of focusing on web application security risks, it tracks and ranks the most common insecurities seen in LLM applications. The most relevant to our research is LLM02: Insecure Output Handling, which occurs when the output of an LLM is not validated and allowed to influence downstream task. It maps to CWE-79, which is one of the weaknesses used in our case study [33]. This underscores the recent prevalence of LLM security concerns.

NIST The **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST) is an agency of the U.S. Department of Commerce which provides measurement, standards, and technology to various American products and services [34]. One focus of NIST is advancement of IT services, which necessarily includes advancement in cybersecurity best practices, standards, and guidelines. A product provided by NIST enabling automation of vulnerability detection and mitigation is the national vulnerability database (NVD). The NVD is a U.S. government-run database chronicling known vulnerabilities and their corresponding fixes containing data of known vulnerabilities in real-world applications, links to the vulnerable GitHub files, and links to the fixing commits (if existent). These data and their CWE mappings are obtainable in a number of ways including API and JSON feeds [35].

MITRE The MITRE Corporation is a nonprofit that manages six Federally Funded Research and Development Centers (FFRDCs), including the National Cybersecurity FFRDC. Two of MITRE's cybersecurity initiatives that have come from this center are the **C**ommon **V**ulnerabilities and **E**xposures (CVE) and **C**ommon **W**eakness **E**numeration (CWE) frameworks. Both initiatives are sponsored by a number of governmental agencies [36].

As discussed at the beginning of this chapter, code with weaknesses which is deployed to production might become exploitable by an end user and is thus vulnerable. If this vulnerability comes to fruition, then its specific instance is logged as a CVE; the original underlying weakness is a CWE. Since CVEs are specific to the vulnerability instance, not every CWE has a corresponding CVE. Both CWEs and CVEs identify and assign numerical IDs to insecure code; the difference is CVE catalogs instances of vulnerabilities and CWE serves as a common language for discussing security risks and mitigation efforts. The CWE initiative is also used as a litmus test for assessing the scope and quality of new security tools [36].

3.3 CWE Deep Dive

Our research uses the CWE system to categorize insecure code. Not only is this industry standard, it is better to target the problem at the source - the pre-production weakness level. In our case study, we investigate rationales for CWE-79 and CWE-89 code samples. While these were the most frequent CWEs in our data, they are also particularly prudent choices. Both are members of Category 137 (Data Neutralization Issues), both have been in the CWE Top 25 for the past 5 years, and both have been named one of the CWE Top 25 Most Dangerous Software Weaknesses of 2023 [37, 38, 39]. Given how frequent these weaknesses are, they are more likely to appear in training data. Moreover, deployment of code with these weaknesses has dire implications, so identifying generated code containing CWE-79 or CWE-89 has a higher priority than other CWEs with less dire consequences. Therefore CWE-79 and CWE-89 provide a good starting place for investigating LLM-generated insecurities.

3.3.1 CWE-79

Improper Neutralization of Input During Web Page Generation, or Cross-Site Scripting (XSS), occurs when a web page implements user input without proper neutralization in such a manner that impacts other end users. The overall process is as follows: first, a web application collects data from an end user, then dynamically generates a webpage containing the data without ensuring the data is acting within its security policy does not contain browser-executable content. When other end users access the page, their browser executes the injected script. Since this malicious script exists in a different domain than the end user, XSS violates the web browser security policy of same-origin [40].

The consequences of XSS attacks range in severity from having your Instagram hijacked to completely losing control over your computer and all information on it. XSS is also extremely common - for example, we have all experienced some form of phishing. For these reasons, CWE-79 has been named as the second most dangerous software weakness of 2023 [40, 39]. XSS can happen in one of three ways.

Type 0

Type 0 is Document Object Model (DOM)-based XSS, where the injection is the result of a server script processing user input and placing it into the original webpage without proper neutralization. Because DOM does not involve any servers, Type 0 exclusively concerns the client side. In other words, the client can change how a webpage looks or acts in their browser by altering the DOM environment. Given the source of the injection and lack of server-side involvement, it is very different from the other two types.

For example, consider this HTML line which controls the color of a banner on a webpage through its URL:

```
document.write('<style>body\{color:'+
    document.URL.substring(pos,document.URL.length)+';\}
```

The URL displayed ends in ”profile?color=blue”. If the malicious actor inspects the webpage and finds the URL is wrapped in a style attribute, they could replace the last part of the url with the following:

```
profile?color=<style><script>alert("Execution successful")</script>
```

in which case the script will execute and display an alert for the user stating ”Execution successful” [41].[1]

Type 1

Reflected XSS occurs when the end user provides dangerous content to a vulnerable web application. Instead of storing said data, the server dynamically reflects the data back to the end user's browser in a HTTP response [40]. A classic example is phishing, which works as follows: you receive a message from a seemingly reliable source - a bank, family member, etc - asking you to click on a link or provide personal information of some sort in a time-sensitive, urgent manner. If you do click on the link, any number of malicious programs could be downloaded onto your device; if you do provide information, your information is now compromised [42].

[1]A fantastic example of this is the result of Twitch connoisseurs discovering a vulnerability during a livestream. A video of the ensuing saga is available here: https://www.youtube.com/watch?v=2GtbY1XWGlQ

Type 2

Type 2 is Stored XSS, also known as HTML injection. The injection occurs when the user provides dangerous data that is stored; thus, the weakness persists in the system until served back to the client. Take, for example, the following SQL code which adds a user's information to a website's database:

```
Insert Into users (username,password) Values ("\%s","\%s","\%s")',
$username, $password, $fullName
```

where username and password are gained via user input, and the website has a page displaying all usernames of active users. Now imagine the user inputs an HTML command as their username, and no sanitization occurs between gaining the input and placing it into the database. When anyone visits the active users page, the HTML contained in the username is executed [40].

3.3.2 CWE-89

Improper Neutralization of Special Elements in a SQL Command (SQL Injection) occurs when SQL commands are built from user input without proper neutralization to ensure the original command will not be altered by the end user's input. The most common way this arises is improper use of single quotation marks in SQL syntax with user input, which enables the end user to inject additional SQL code into the query [43].

For example, let us consider the following SQL query:

```
"SELECT * FROM items
WHERE owner = '" + userName + "'
AND itemname = '" + ItemName.Text + "'"
```

Suppose ItemName is a field obtained via user input, no input neutralization is applied before creating the query, and no neutralization is applied before executing the query. If the user were to provide the system with input including single quotation marks like so:

```
x OR 'a'='a
```

then the query will select every entry from items as the OR statement, which is always true, has made the WHERE clause perpetually true as well [43].

The consequences of this weakness are dire: the ability of an end user to not only access sensitive information but potentially change it. Either the injected SQL grants the end user access to other accounts without a password, or the injected SQL grants the end user the ability to change other users' passwords. Furthermore, user bases are commonly subject to SQL injection attacks because of how easy it is not only to notice but also to exploit [43]. For these reasons, CWE-89 has been named the third most dangerous software weakness of 2023 [39]. Additionally, this weakness has steadily climbed the ranks of the Top 25 list from 6th to 3rd place over the past few years [44].

Chapter 4

Language Modeling for Code Generation

Code generation, particularly through use of neural code models (NCMs), is a prominent research area in software engineering (SE). NCMs often leverage large language models (LLMs) as the vehicle for code generation; by learning representations of input sequences, such NCMs are able to generate similar sequences provided a history. If the input sequences are code, the generated sequences will also be code. Thus NCMs are applicable to any number of SE tasks including but not limited to code completion, program repair, and test case generation [21].

4.1 Language Modeling

The end goal of a language model is to learn a representation of some input sequence. Specific domains of sequence based data are called corpora and can be transformed into a sequence of discrete objects through a transformation function. The resulting tokenized sequence $w_{1:T}$ has conditions $1 \leq t \leq T$, a granularity defined by the transformation function, and $w_t \in V$ where V is the set of all possible tokens - our vocabulary. For SE LM tasks, our corpora is the software corpora, which is composed of software artifacts and our tokenizer can decompose the software corpora into tokens, words, or sub-words [23]. We must first represent our data in such a manner before modeling it.

Following from this discrete sequential data representation, statistical LMs are defined by the following:

$$P_\theta(S) = P_\theta(w_{1:T}) = \Pi_{t=1}^{T} P_\theta(w_t|w_{<t}) \tag{4.1}$$

where P_θ is a probability distribution, S is a sequence with fixed granularity obtained from our corpora C, and $\Pi_{t=1}^{T} P_\theta(w_t|w_{<t})$ is a conditional distribution of w_t provided our input sequence $w_{<t}$. Typically this conditional distribution

is estimated by a neural language model (NLM) with classification abilities. As such, $P_\theta(w_t|w_{<t})$ is actually $P(w_t|h_t)$: the probability of token w_t provided some hidden state h_t [23]. This latent learning approach relies on the fact that information about input sequence $w_{<t}$ has been embedded in h_t. The process by which this information persists in the hidden state depends on how the model updates the hidden state. Either it is updated with the full sequence of previous inputs $w_{1:t-1}$ and the previous hidden state, or with just the current input w_t and the previous hidden state. The former is the autoregressive approach and the latter is the recurrent neural network approach [45].

We use the autoregressive approach (specifically transformers) because such models are able to model long-term dependencies in the input sequence [45]. Due to this property they are often used for sequence generation tasks, where the model generates some $\hat{w_t}$ from $P(w_t|w_{<t})$; since this generated token is conditioned by the input sequence, our generated sequence maintains legibility when compared to the input sequence as it increases in tokens over time. This probability is often practically represented by the softmax function, which produces the distribution of probabilities over all tokens in our vocabulary. It is influenced by cross-entropy loss, which relates the probability generated by the hidden state to the distribution of our truth $P(w_t|w_{<t})$ [23]. The cross-entropy loss guides model learning through providing a metric of model performance; as the model learns, it changes its weights to minimize this loss, thereby altering h_t and $P(w_t|h_t)$ [46].

4.2 Finetuning

Finetuning is a method for making a trained model attuned to a specific task [47]. This method continues the training and changes model parameters through introduction of differently structured data [48] and is described as follows:

$$P_\theta(S) = P_\theta(w_{1:T}) = F_\theta(w_1)\Pi_{t=2}^{T} F_\theta(w_t|w_{<t}) \tag{4.2}$$

where all variables are defined as in Eq. 4.1 except for F_θ, which is the trained model we are fine-tuning. By comparison with Eq. 4.1 we can see that the fine-tuning process is effectively training scaled by $F_\theta(w_1)$ to maintain some amount of original performance, thereby preventing model weights from changing drastically.

Chapter 5

Interpretability

Interpretability analysis *interprets* the learned relationship between model weights and input sequences through providing explanations of model outputs [21]. As discussed in the previous chapter, NLM outputs can be described as the conditional probability $P(w_t|h_t)$. These probabilities exist in model space $\vec{\ell}$ and are generated based on the model weights and hidden state. Herein lies the interpretability challenge: instead of existing in $\vec{\ell}$ space, humans exist in $\vec{h}$ space, whose domain consists of human-interpretable concepts $\mathcal{H}$. In other words, hidden states and weights are inherently interpretable to the model but not to humans. To solve this problem we apply a post-hoc function for interpretability ϕ mapping $\vec{\ell}$ to $\vec{h}$ [23], after which we are able to interpret $P(w_t|h_t)$ in terms of human-interpretable concepts $\mathcal{H}$.

This application can occur on both local and global levels. Locally, we apply ϕ as such: $\phi(S, \mathcal{H})$, where S is one input sequence. If, however, we would like to investigate a model across many input sequences, then we require additional steps. Thus we define a function g which aggregates $\phi(S, \mathcal{H})$ over n sample sequences. The entire global process can be described by:

$$\Phi(g, S^n, \mathcal{H}) = g(\phi_{\mathcal{H}}, S^n) = \frac{1}{n}\sum_{j=1}^{n} \phi_{\mathcal{H}}(S^j, \mathcal{H}) \tag{5.1}$$

where our aggregation function g is defined as expected value, and the resultant explanation can be targeted toward a specific human-interpretable concept. These concepts are defined by the specific task and intended analysis [21]. See Sec. 6.2 for a discussion on human-interpretable concepts in code.

5.1 Rationales

One such ϕ is sequential rationales, which was first proposed by Vafa et al. in 2021. Recall Ch. 4 and in particular Eq. 4.1, which describes language modeling through some probabilistic model P_θ and sequence $w_{1:T}$. We can utilize said model to generate sequences through repeated generation of individual tokens w_t as a conditioned by their context window $w_{<t}$. Any subset of the context window $w_{<t}$ that leads to P_θ predicting w_t is called a *rationale*. The context itself is always a rationale by definition; however, it does not provide us with any additional insight to model predictions and can be difficult to interpret. Our goal is thus to find the minimum combination of tokens in the context that result in model production of the generated token. This is known as the optimal rationale; since at some point predicting w_t depends on the optimal rationale, we can express the conditions of this rationale through combinatorial optimization as follows:

$$argmin_{r_w \in \mathcal{R}} |r| : argmax_{w_t^{'}} P_\theta(w_t^{'} | w_{r_w}) = w_t \tag{5.2}$$

where $\mathcal{R} = 2^{[t-1]}$ is the power set of all possible subsets of $w_{1:T}$, w_t is the generated token, P_θ is our model, and $P_\theta(w_t^{'}|w_{r_w})$ is the probability of generating token $w_t^{'}$ given a rationale w_{r_w}. The *argmin* side of Eq. 5.2 ensures our objective of a short rationale and the *argmax* side ensures our constraint, namely that the subset of the context $w_{<t}$ which we obtain is a valid rationale. However, optimizing our rationale for interpretability purposes is only one third of the challenge: actually solving Eq. 5.2 is N.P. hard, and model predictions depend on the full context $w_{<t}$ while rationales depend on arbitrary subsets of the context which are shorter than the full context and thus out of sample for the model. To resolve these issues, we can approximate the equation's solution through the greedy rationalization algorithm and finetune the model for compatibility with inputs of varying length.

Greedy Rationalization Greedy rationalization is a greedy algorithm which builds the optimal rationale as follows: First, start with the empty set. Then calculate the conditional probability of w_t given each token in the candidate rationale context window $w_{<t}$. The token in $w_{<t}$ which has the largest probability of producing w_t is then added to the set; once the conditional probability of the set produces the given token, the algorithm halts. In the worst case scenario we arrive at the full rationale, thus the algorithm will always converge.

Compatibility During the training phase, most models train on the entire context so the subsets required for intermediate calculation steps are out of sample. Unless we train on subsets of the context, we can't generalize to it. We can fix this problem easily through finetuning with data of varying context lengths. This is realized through using incomplete contexts w_r from power set $\mathcal{R}$ for use as inputs to trained model F_θ. Since F_θ is already trained, it is still able to predict a token w_t given its full context $w_{<t}$ once compatibalized [12].

Chapter 6

Our Implementation

6.1 The Model

To implement sequential rationales we need not only data but also a model for Python code generation which we can finetune for compatibalization. We chose CodeParrot, an open-source GPT-2 model trained from scratch exclusively for Python code generation capabilities [49]. It uses a custom tokenizer for Python code, and effectively has the functionality of ChatGPT but exclusively for Python code [50]. We used the small version, which has 110 million paramers and was trained on the cleaned CodeParrot dataset. This dataset contains roughly 50 GB of Python code from GitHub distributed over 5,361,373 files, and is split into training and validation sets[1]. Furthermore, they performed filtering on the data: the dataset only contains code samples whose average line length is between 100 and 1000 characters. Lastly, the model's context size is 1024 tokens [51]. For brevity, general information on the transformer architecture, GPT, and GPT-2 is located in Appendix B.

6.2 Syntax Aggregations for Rationales

As discussed in Ch. 5, we use rationales as our function for interpretability ϕ which can be targeted towards specific human-interpretable concepts. Such concepts exist on the token, token aggregation, token hierarchy, and token scope levels. This leaves us with a decision to make: which level do we analyze our rationales on? We propose five possible human-interpretable concepts for code rationales: $\mathcal{H}^{(0)}$, code rationales without aggregation; $\mathcal{H}^{(1)}$, structural code taxonomy; $\mathcal{H}^{(2)}$, identifier concepts; $\mathcal{H}^{(3)}$, code context scopes; and $\mathcal{H}^{(4)}$, natural language based scopes. Of these five, we use $\mathcal{H}^{(0)}$ and $\mathcal{H}^{(1)}$ for our results.

All concepts have varying levels of aggregation, which relies on the ability to parse rationales and map them to some corresponding higher-level concept or

[1]The split is roughly 87% training and 13% validation.

role with semantic meaning in the given block of code, at which point we are able to aggregate the rationale values and attribute them to the mapped concept. Differences in this mapping define our $\mathcal{H}$ proposals. In other words, mapping rationales to their semantic meaning effectively categorizes the rationales with varying levels of granularity. This results in the ability to explain what parts of the context result in vulnerabilities [21]. Our two concepts of choice, $\mathcal{H}^{(0)}$ and $\mathcal{H}^{(1)}$, provide us with the ability to extract explanations for insecure code generation. $\mathcal{H}^{(0)}$ is the rationale itself, and as such has already been discussed in the previous chapter. $\mathcal{H}^{(1)}$ categorizes parts of code into mappable structural concepts such as comments or loops, and aggregation is carried out via statistical functions. This provides us with a way to investigate rationales in terms of PL code concepts interpretable to developers: a structural code taxonomy [21].

6.2.1 Structural Code Taxonomy

Effectively what we are doing is grouping tokens by PL concepts through parsing the code's AST representation and aggregating the rationale values by node type mapping [21]. The structural code taxonomy can be divided into two main sections, programming language (code) and natural language. Both of these can be further divided into more categories. For programming language, tokens can be semantic, natural language in code, syntax errors, non-semantic, or a context window. Semantic tokens in programming languages can be data types, e.g. float or char; exceptions, e.g. try or catch; oop, e.g. public or new; conditionals, e.g. if/else; loops, e.g. for/while; bool, e.g. Boolean; with, e.g. with clause; structural, e.g. attribute or module; asserts, which is only assert; and statements, which is only assignment. Natural language in code includes identifiers, comments, and strings - anything that is natural language contained in code. There is also a category for tokens which are syntax errors. Non-semantic programming language tokens can be expressions, e.g. call or async; punctuation; operators, e.g. in or %; indentation, namely '\n' or '\t'; return; or functional. Lastly, programming language tokens can be part of a context window: class, focal method, constructor, signature, or field. Natural language tokens can be semantic or non-semantic. Semantic natural language tokens can be verbs, nouns, or pronouns; non-semantic natural language tokens can be prepositions, determiners, adverbs, conjunctions, cardinals, particles, modals, or lists.

Chapter 7

The Data

All machine learning starts with the most important element: data. To apply greedy rationalization and draw useful interpretations from it, we require insecure code samples with their CWE mapping and vulnerability span at bare minimum. Without the code samples, we cannot generate rationales; without the vulnerability span, we cannot generate rationales which lead to insecure code generation; and without the CWE mapping, we cannot explore the extent to which rationales depend on the CWE. There were two main options for this study: one, create the data by hand which would be extremely time-consuming and is thus undesirable; or two, find a dataset that already exists. In our pursuit of option two, we discovered that there is a chronic shortage of datasets that have vulnerable/secure Python code pairs with corresponding vulnerability spans. Thus we pursued Option 1.5: Combine existing datasets requiring minimal amounts of manual analysis, thereby maximizing the amount of data in a minimal amount of time. Our data is sourced from three existing datasets: SecurityEval, CWE Scenarios, and CVEFixes.

In this chapter we will discuss how our data is organized; the data collection methodology our three sources use, and how we reorganized them; our full dataset; and the data used for this case study. We will end the chapter with exploratory data analysis of our full and case study datasets. Additional exploratory data analysis of the three source datasets can be found in Appendix A.

7.1 Organization

Combining the datasets necessitates placing each one into a common format and system of organization beneficial for our study. Therefore, we reorganized each source dataset into individual JSON objects and then combined those JSON objects to obtain our full dataset. Each JSON object contains fifteen fields, seven of which are calculated based on the specific vulnerable code sample.

7.1.1 Abstract Syntax Trees

Our calculated fields heavily depend on abstract syntax tree (AST) representations of code. An abstract syntax tree is a hierarchical graph representation of source code typically used in coding language compilation [52]. This representation stores all syntactic qualities of the original code in a tree structure with nodes [23]. By parsing the tree with Tree-Sitter [53] we are able to extract useful information about the source code including but not limited to the node type, number of AST errors, the AST height, and number of nodes.

From AST properties, we can calculate particular code metrics. One such metric is cyclomatic complexity, which measures the complexity of conditional logic in a block of code. The higher the complexity, the higher the score, and the lower the efficiency: cyclomatic complexity effectively measures all possible paths through a block of code by summing the number of logical forks in the road Using Radon's documentation as a guide, we calculated the cyclomatic complexity of vulnerable code samples by iterating through the sample's detected AST node types and adding one each time a node type introduced a fork in the road [54].

7.1.2 Fields

As one might expect, the first field is a numerical ID for each entry in the JSON object. Each file's ID is based on the length of the file. The second is the corresponding processed insecure code sample; for our first two sources, this field is the insecure code generated by the model. Third is the corresponding secure code or prompt, which is not processed. Field four is the sample's CWE and field five is the vulnerability span. For identification purposes, fields six and seven are the source dataset (SecEval, Copilot, or CVEFixes) and the language of the code samples, which is always Python. Field eight contains a short description of the particular CWE obtained from the MITRE definition. From the vulnerable code's abstract syntax trees we obtain and store the number of AST errors, nodes, and the tree height in fields 9, 12, and 15, respectively. In addition to this information, we also calculate the vulnerable code's number of whitespaces, lines of code, and words. These are stored in fields 10, 11, and 12, respectively. Lastly, we calculate the cyclo complexity for the vulnerable code which is then placed in field fourteen.

We chose these specific measures for several reasons. First, we can investigate the distribution of each metric by dataset which enables cross-dataset and cross-CWE code sample comparison. Second, each metric is relevant to measuring different aspects of the code. By investigating the code's AST representation and other metrics such as lines of code, we are able to obtain a well-rounded evaluation of not only the complexity of a sample but also its use of space and hierarchical structure. Furthermore, when we calculate rationales we define an aggregation function which directly relates to a hierarchical representation of code. See Ch. 6 for more on aggregation functions. The idea is by investigating these properties of the vulnerable code and comparing to rationales, we might be able to explain or identify whether certain metric values are more likely to result in insecure code generation. This capability was not necessary for this study; as such, our calculated fields are only relevant for exploratory data analysis at this point in time.

The vulnerability span is the most critical field in our data. This is the span of vulnerable tokens in the code, starting with the first vulnerable token and ending with the last vulnerable token in the format:

[[(row1,col1),(row2,col2)],[(row1,col1),(row2,col2)]]

and so on, with two row-column pairs per vulnerable token span. Without this information, we are unable to generate rationales for the weakness and therefore are unable to draw any meaningful conclusions from the rationales. Just as there is a chronic lack of data, there is also a chronic lack of reliable and accurate tools for automatic vulnerability span extraction. All 121 samples from the SecEval dataset were manually reviewed to obtain the vulnerability span and all CVEFixes samples used in our case study were also manually reviewed. The Copilot dataset uses a tool called CodeQL to detect vulnerabilities and extract the vulnerability span.

7.2 Our Sources

7.2.1 SecurityEval Dataset

The SecurityEval dataset was created to help developers **eval**uate the **security** of LLM-generated code by Siddiq and Santos in 2022 [55]. The dataset consists of 121 pairs of LLM prompt/generated insecure Python code pairs and covers 61 CWEs. Their data collection process is similar to what we have done: they mined samples of vulnerable code with a CWE mapping from 4 sources, obtaining 36 samples from CodeQL, 11 samples from the CWE website, 34 samples from SonarRules, and 4 samples from the CWE Scenarios dataset. To round out the number of CWEs represented in the dataset and increase the number of prompt/insecure code pairs, they created the remaining 45 samples.

CodeQL is a tool created by GitHub for identifying and inspecting vulnerabilities in source code [56]. The samples in SecEval are taken from the CodeQL documentation. SonarSource is a company that provides a static code analysis tool, SonarRules [57]. These rules include CWEs and the documentation provides sample insecure and secure code pairs, which were extracted and consequently used for SecEval. Lastly, the CWE Scenarios dataset was released as part of Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions (Pearce et al., 2021). As this is one of our dataset sources, we leave discussion to the next section.

This dataset aligns *almost* perfectly with our research goals; the only missing field is vulnerability span. Since the dataset is available on their Github repository as a JSON file[1], we exported the data to Excel for manual review. Our

[1]Available at https://github.com/s2e-lab/SecurityEval/blob/main/dataset.jsonl

manual review was for quality and vulnerability span extraction. We denoted the vulnerability span in the source code itself by placing an asterisk before the first vulnerable token and after the last vulnerable token. By doing this, we were able to extract the vulnerability span by locating the asterisks. Samples which include SQL may contain an asterisk, so for these samples we used an exclamation mark as our delimiter instead. For simplicity we refer to both delimiters as asterisks in this thesis. We only kept samples that had a clear vulnerability span.

Reorganization Process and Vulnerability Span The bulk of our reorganization process is ensuring the vulnerability span was extracted correctly, and adjusting the vulnerability span after code cleaning. The process is as follows: After loading in the Excel file, we iterate through each row. At every row, we first check if the code sample has an asterisk - if it does, we continue with reorganization, and if it doesn't, we move to the next row. Next, we split the code into a list of lines and filter out empty lines and lines only containing comments. Then we obtain the vulnerability span by iterating through the lines of code until we reach a line with an asterisk. Once at this line, we iterate through tokens in the line until we reach the asterisk location. If our line iterator is "i" and our token iterator is "j", then the start of the vulnerability happens at row i and column j. This (i,j) is appended to our vulnerability span list. We keep iterating through the tokens as before until we reach the second asterisk, which may be on the next line. Once we have reached the second asterisk, we append (i,j) to our vulnerability span list and return the list. We then filter out the asterisk. This requires adjusting the vulnerability span by subtracting one from the second column. To ensure this worked, we check that the new vulnerability span returns the same tokens as the original span with the unfiltered code. If it is the same, we recombine the list of lines into a string and inspect it to obtain our calculated fields. The SecEval dataset contains the ownership information in its ID field; we give the samples an ID corresponding to the length of the list and simply call the source SecEval instead. Lastly, we obtain the CWE, prompt, and description from the original file. We combine these into a dictionary and append the dictionary to a list.

After reorganizing and cleaning the data, we had 72 prompts for 39 CWEs. While the content of the data is fantastic for our purposes, there were only one or two samples per CWE. Thus the data was too broad in CWE scope and we would not be able to generalize our results. We needed more data.

7.2.2 CWE Scenarios Dataset

Copilot is GitHub's AI assistant programmer, which can autofill and complete code for users. The tool employs a proprietary code-scanning process to extract context from the existing code, which is then used as the input to a code generating LLM. For all intents and purposes this is effectively an auto-generated prompt created from preexisting code written by the programmer. Since the LLM Copilot relies on is trained on a large dataset of real-world code which might contain bugs or insecurities, there are concerns around the quality and security of Copilot-generated code [9].

Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions (Pearce et al, 2021) conducts an empirical security evaluation of Copilot-generated code. They used GitHub's security-oriented semantic code analysis tool, CodeQL, to detect insecurities. This tool utilizes its unique QL language to build and query code databases; since there are pre-written queries for many CWEs, it is a convenient way to detect insecurities across many code samples. CodeQL results are stored in a CSV file which contains the detected CWE ID, vulnerability span, the filename which contains the weak code, and the CWE's short description from the MITRE website.

Pearce et al.'s evaluation was carried out as follows. First the authors gathered scenarios, which serve as the context for Copilot-powered code completion. These scenarios are incomplete in such a manner that can easily lead to an insecurity, and were gathered from 1) the CodeQL documentation; 2) the MITRE CWE database; or 3) handwritten by the authors. Copilot then completes the code in up to 25 ways. These resultant code blocks are then analyzed by CodeQL to determine whether they contain the intended insecurity. Samples which CodeQL was not able to evaluate (e.g., the generated code would not compile) were manually evaluated by the authors. While the scenarios are intended to generate a specific insecurity, it is always possible for Copilot to generate code with some other, unintended weakness. As generated code samples were only evaluated for their intended weakness, lack of identified weakness does not ensure the generated code was secure. With this being said, 40% of the generated code samples contained the intended insecurity.

This evaluation was carried out through three experiments: **d**iversity **o**f **p**rompt (DOP), **d**iversity **o**f **w**eakness (DOW), and **d**iversity **o**f **d**omain (DOD). DOD investigates whether the PL impacts vulnerability frequency in generated code; our study uses Python code exclusively, so this experiment is wholly irrelevant to our purposes and we only extracted data from the other two experiments. DOW investigates whether Copilot is more prone to generating certain types of weaknesses through a variety of scenarios based on the MITRE Top 25. DOP investigates the impact certain elements of code context have on Copilot generation of CWE-89. The DOP scenario premise we investigated was asking Copilot to complete code allowing a user to unsubscribe to a service.

The results of these experiments are contained in a dataset available on Zenodo[2]. It contains 1,189 programs across 89 CWEs. These data are organized into three folders corresponding to the study's three experiments. To extract data from the DOP and DOW experiments, we had to parse the file system, and in doing so extracted the insecure code and its corresponding scenario, CWE information, and vulnerability span.

There are a few preliminary qualifications to identifying usable data. First, not all programs resulted in insecurities; in these cases, the corresponding CodeQL CSV file was empty and we did not use the code samples. Second, we

[2]Available at https://zenodo.org/records/5225651 or https://zenodo.org/records/6975069

exclusively extracted data where the program was written in Python. Third, CodeQL results provide the name of the file containing the detected weakness but not the filepath itself. While this is not an issue with the DOW setup, the DOP setup is more complicated and thus finding the correct filepath can be challenging. There were 15 files containing CodeQL-detected insecurities that we could not find in the original data - these are not included in our dataset, but they could be manually located and added in the future.

Filesystem Details The DOP results are located in experiments_dop/cwe-89/unsubscribe. This folder has a number of CSV files containing CodeQL results for individual scenarios as well as the original Python scenarios before Copilot completion. The CodeQL results relate to a generated Python file containing an instance of CWE-89, and are located in folders starting with "gen_" which contain all samples completed by CodeQL. The DOW results are located in the experiments_dow folder. Since each DOW scenario is specific to a particular CWE in the MITRE Top 25[3], this folder is further organized into folders for each CWE. Within each of these CWE folders are folders for the results of particular scenarios; lastly, within these scenario folders is the scenario file, CodeQL results in CSV format, and a folder containing all Copilot-completed code samples. Not all scenarios are written in Python.

Reorganization Process Our reorganization process for the DOW experiment is as follows: For each folder in the DOW folder, obtain the CWE from the folder's name. Then for each experiment contained in the CWE folder, obtain the scenario file. If this file is not a Python file, we ignore the sample and continue on. If the file is a Python file, read it as a string to extract the scenario. Next, we locate the CodeQL results and read the CSV into a DataFrame. Sometimes, there will be no such file, the file is empty, or the DataFrame is empty - in these cases, we ignore the scenario and continue on. For each row in the DataFrame, we obtain: the CWE description, vulnerability span, and filename containing the insecurity. From the filename we create the filepath and read in the insecure code. Then we remove empty lines and extraneous comments from the insecure code. Lastly, we obtain our calculated fields and append the dictionary to our list, where the sample's ID is the current length of the list. The DOP experiment reorganization process only differs in file traversal, and the fact that there are only samples for CWE-89.

Vulnerability Span Difficulties The main challenge with reorganization was vulnerability span extraction. While obtaining the span detected by CodeQL was simple, most of the time this span was not accurate to the location of the vulnerability. More specifically, the indicated row(s) were incorrect. This was discovered because the column index would be greater than the length of the specified line of code in some cases. Thus, we had to add functionality to

[3]CWEs 20, 22, 78, 79, 89, 119, 125, 190, 200, 306, 416, 434, 476, 502, 522, 732, 787, and 798

relocate the correct row and fix the span. We did this by iterating through the lines of code until the length of the line was greater than or equal to the column value. Once we had found the correct row, we updated the span and confirmed it updated correctly by comparing to the original manually adjusted span.

Notes on code cleaning On investigation of the insecure code samples themselves, we noticed there was additional extraneous information contained in comments that should not be included in calculation of rationales as they were not used by Copilot for code completion. These are authorship attribution and Copilot mean probability scores. Thus we filtered these out in addition to empty lines.

After extracting, cleaning, and consolidating the data from both experiments we had 285 samples for 5 CWEs, the vast majority of which were samples of CWE-89 likely sourced from the DOP experiment. This data and the SecEval data were reorganized in the same notebook; thus once we had completed reorganizing the CWE Scenarios dataset we had completed reorganizing our data and consequently saved these two combined datasets to a JSON file.

7.2.3 CVEFixes

Our study exclusively investigates insecure code generation; however, it could also be useful to investigate the differences between secure and insecure code generation. For this we need pairs of insecure and secure code instead of prompts and generated code. We can still apply greedy rationalization on secure/insecure code pairs by treating the sequence of code tokens before the insecurity as the prompt.

CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software (Bhandari et al, 2021) introduces a method of obtaining such code pairs by leveraging the national vulnerability database (NVD) in concert with GitHub repository scraping. CVEFixes extracts vulnerability and corresponding fix data from the NVD JSON vulnerability feeds, which are updated daily. From these feeds we can obtain the mapping between CWE and CVE, the observed vulnerability, a link to the corresponding GitHub repository, and a link to the associated fixing commit (if existent). The links are then used to scrape the GitHub repository on both the file and method levels to extract more commit information and the corresponding code for the vulnerability and fix. CVEFixes only collects vulnerabilities with fixes [58].

The collected data is then placed into a SQL database with 8 tables. Three tables are dedicated to CVE and CWE information, both in general and in relation to the particular vulnerable code. One table connects the three CVE/CWE tables with the remaining four, which are dedicated to information about the source repository and both method and file level changes to the code on a commit basis. From this database we can obtain the CWE, its description, and the programming language. We then have to extract our vulnerable and secure code samples from the method and file level tables. Some metrics such as number of lines of code and complexity are included in the database. However, because we employ additional code processing such as removing comments and empty lines, we must create all of our calculated fields.

There are two ways to use CVEFixes: recreate the database from a SQL dump of the initial release created in 2021, or create the database from scratch. The initial release did not contain enough Python samples, so we recreated the database from scratch. We utilized CVEFixes to collect 11,997 samples covering 9,148 CVEs which correspond to 263 CWEs. Of these, there are only 2,122 Python samples covering 778 CVEs which correspond to 116 CWEs.

Reorganization Process The reorganization process is as follows: We began by creating a view with the CWE name, id, and description from the CWE table, CVE id from the CWE Classification table, and hash (ID for a particular code fix) from the Fixes table. This view was grouped by the CVE ID and hash, and only contains entries for named CWEs. Next, we used this view to create another view connecting CWE mappings to method and file changes. This was made possible by first combining the previous view with the File Change table using the hash. We only considered entries written in Python and with a valid file change id. Then we performed a left join with the Method Change table using the file change id. For both views we grouped by CVE and method with granularity in mind - one CWE can map to many CVEs, and one file can have many method changes. Next, we implemented this second view to obtain the insecure and secure code through leveraging the before change Boolean field. We end with two dataframes, both containing the same fields: CWE ID, CWE name, CWE description, filename, code after, code before, file change id, method change id, start line, end line, code, and before change. The insecure code dataframe is where before change is true, and the secure code dataframe is where before change is false. The method before or after fixing the insecurity is contained in the code field. Both are grouped by method change id. From here we combined the two dataframes, obtained the insecure and secure code, filtered out comments and imports on both, obtained our calculated fields, and added to our JSON object.

After completing our extensive reorganization and cleaning process, we have 812 pairs of insecure/secure Python code samples across 86 CWEs. Note that there is no vulnerability span associated with any of these samples. Manual review of this many samples would be extremely time-consuming; thus we leave it as is for later extraction.

7.3 Full Dataset

Once we had cleaned and placed each dataset into a common format, it was child's play to combine them into one JSON object - the only field that qualitatively changed was the ID number, and the name of the SecEval/Copilot dataset's code sample fields changed from source_code to insecure_code and from prompt to fixed_code. Our full dataset consists of 1,169 vulnerable code samples distributed over 117 CWEs. It is important to note that the majority[4] of samples are from the CVEFixes dataset, which does not have vulnerability span information; thus, the dataset is currently incomplete.

7.4 Case Study Data

When determining how to investigate the rationales of our data, it quickly became apparent that an in-depth study of the rationales for a small number of weaknesses would be most beneficial. Due to the limited nature of our data, conducting a case study on the two CWEs with the most code samples is the best approach as it can provide us with insight on whether rationales for the two CWEs are similar or different. As seen in Fig. 7.3, the full dataset is dominated by samples of CWE-89. Given its salience in real-world code and dire consequences as discussed in Sec. 3.3, CWE-89 is a prudent choice for an inital analysis. The CWE with the second highest number of samples is CWE-79. Equally as frequent and consequential as CWE-89, its weak code is very different than CWE-89's as it involves web processes while CWE-89 is defined by SQL use. Therefore CWE-79 is a clear choice for our second weakness of interest, particularly since it also arises from improper input processing.

To shorten the greedy rationalization algorithm runtime we conducted more processing on the insecure code samples to further limit the number of input tokens. First, we removed all tokens following the last vulnerable token; since these are after the vulnerability, they are irrelevant for vulnerable token generation. Second, we removed all imports and comments. This necessitated vulnerability span adjustment and code metric recalculation.

The case study data is not simply the subset of our full dataset containing CWE-89 and CWE-79 samples. Notice that the case study data includes data from CVEFixes, which does not have vulnerability spans. This was remedied by an expedited manual review process of the original 99 CVEFixes code samples of our two CWEs. For some samples, the location of the vulnerability was obfuscated due to high complexity and a drastically different fixing commit; these were removed from the case study data.

Expedited Manual Review for Vulnerability Span Extraction The process is as follows: After removing comments, we identified the potentially insecure lines and their potential fixes through obtaining lines from the insecure code that were not in the secure code, and lines from the secure code that were not in the insecure code. The row location of these candidate insecurities are then added to the vulnerability span. From here, we investigated the potential fixes and vulnerabilities to determine whether they were, in fact, vulnerable. If so, we determined the numerical value of col1 and col2. These were entered as user input and used to complete the span. Some insecurities had more than one span, and some code pairs warranted manual review of the full method. Two

[4]Approximately 76%.

special user input cases in this process are 0 to move on to the next sample, and -1 to make that particular span an empty list which would be deleted later, denoting the sample as secure or too complex.

7.5 Exploratory Data Analysis

Fig. 7.3 displays the distribution of CWE samples in our full dataset by source in order from highest to lowest number of samples. It is clear that our full dataset is overwhelmingly dominated by samples of CWE-89 from the combined SecEval/Copilot source dataset, with CWE-79 as the runner-up. This bar plot also displays the wide spread of CWE samples in our dataset, serving as a visual justification for our decision to choose the two CWEs with the most samples for our case study. Figure 7.4 further investigates this CWE distribution through providing a breakdown of the number of CWEs, total code samples, samples of CWE-79, and samples of CWE-89 for all three source datasets as well as our full dataset and our case study dataset. Similarly, Figure 7.5 provides a breakdown of CWEs in our case study data's CWE through the number of samples contributed by source datasets. The boxplots investigate the distribution of code metric fields in our full dataset and case study data by case study CWEs. Because our dataset has very limited size, we performed bootstrapping on code metric fields.

Bootstrapping is a statistical method enabling generalization of statistics obtained from small sample sizes to larger, more representative sample sizes through resampling with replacement. Through resampling, bootstrapping artificially generates more data based on a statistical function. Once we have resampled our data many times, we can calculate confidence intervals for a descriptive statistical function such as the mean or variance [59]. We used the mean as our function to resample with; we resampled 10,000 times and calculated 97% and 2.5% confidence intervals for the mean. This is true for our exploratory data analysis and our global results analysis.

Comparing the right subplots of Fig. 7.1 and Fig. 7.2, we can see that the mean number of words, number of lines of code, and number of whitespaces are much smaller for the case study dataset than the full dataset. This is expected since our case study dataset involves removing all imports and comments from the code in the full dataset. Comparing the left subplots we observe the opposite trend for AST errors: values for the case study dataset are generally larger than the full dataset in both CWE-79 and 89, indicating our case study data has (on average) more syntax errors than the full dataset. This is also expected, due to the number of samples from CVEFixes in the full dataset as compared to the case study dataset. The AST height and cyclomatic complexity for the case study dataset are slightly lower than the full dataset, which indicates that our case study data is logically similar to the full dataset. AST nodes has the most drastic difference between the datasets; there is a broader range of nodes for CWE-79 in the case study data, and there are much less nodes for CWE-89 in the case study data. This can be explained by the prevalence of CVEFixes data for CWE-89 in the full dataset. From Fig. 7.4 and Fig. 7.5 we can see there are 99 CVEFixes samples in the full dataset for CWE-79 and 89, 49 of which were used for CWE-79 samples in the case study dataset and 7 of which were used for CWE-89 samples in the case study dataset. The 13 CWE-89 samples from CVEFixes excluded from the case study dataset were exceedingly long, had nonsensical fixes, or did not contain an identifiable insecurity. Again, this makes sense; CVEFixes contains real-world insecurities instead of cultivated, short, easily identifiable examples. Overall, the code used for our case study is relatively short (method level), with easily spotted insecurities and relatively simple syntactic structure.

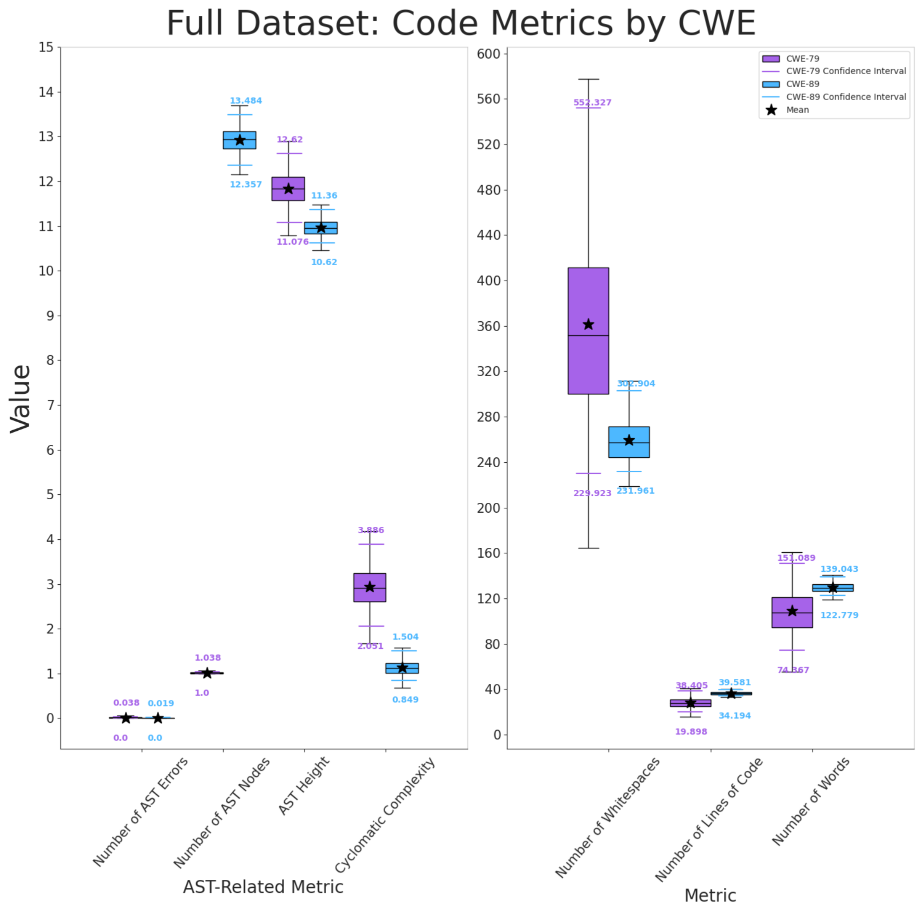

Figure 7.1: Full Dataset by CWE

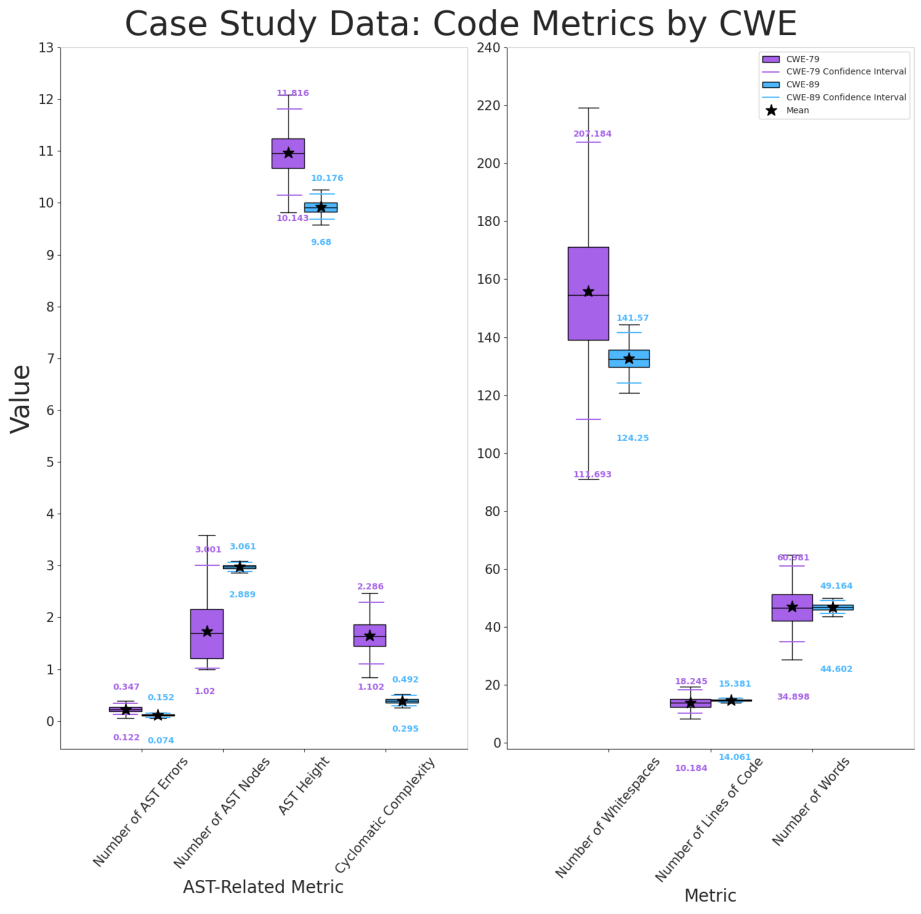

Figure 7.2: Case Study Data by CWE

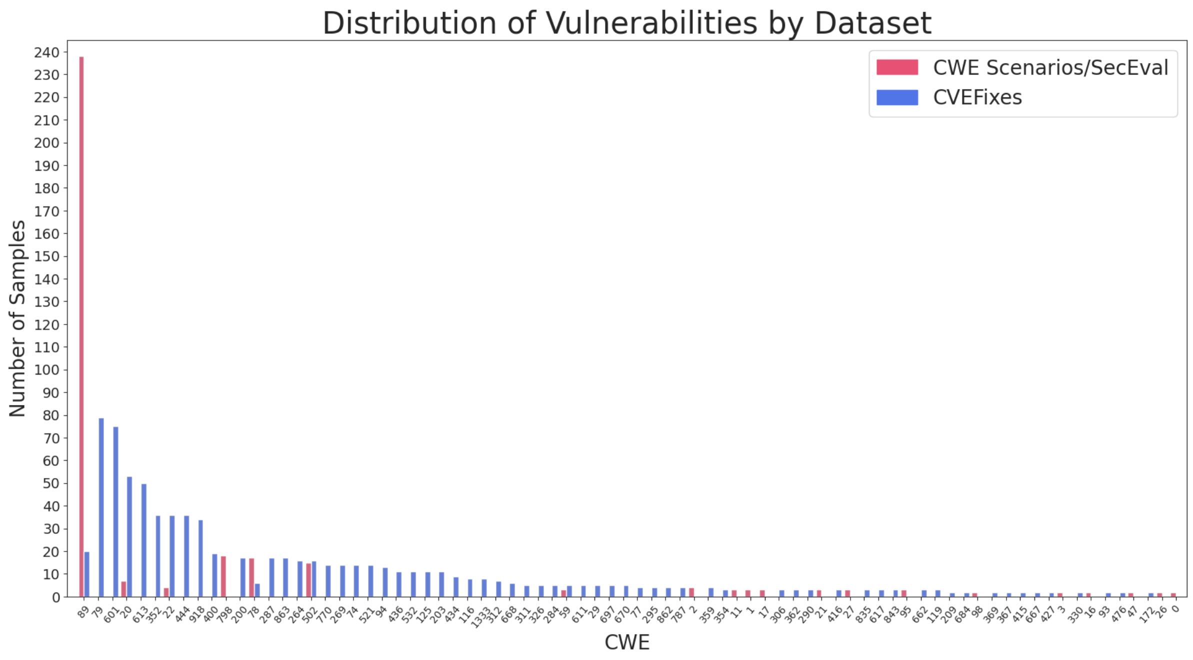

Figure 7.3: Full Dataset: Number of CWE Samples

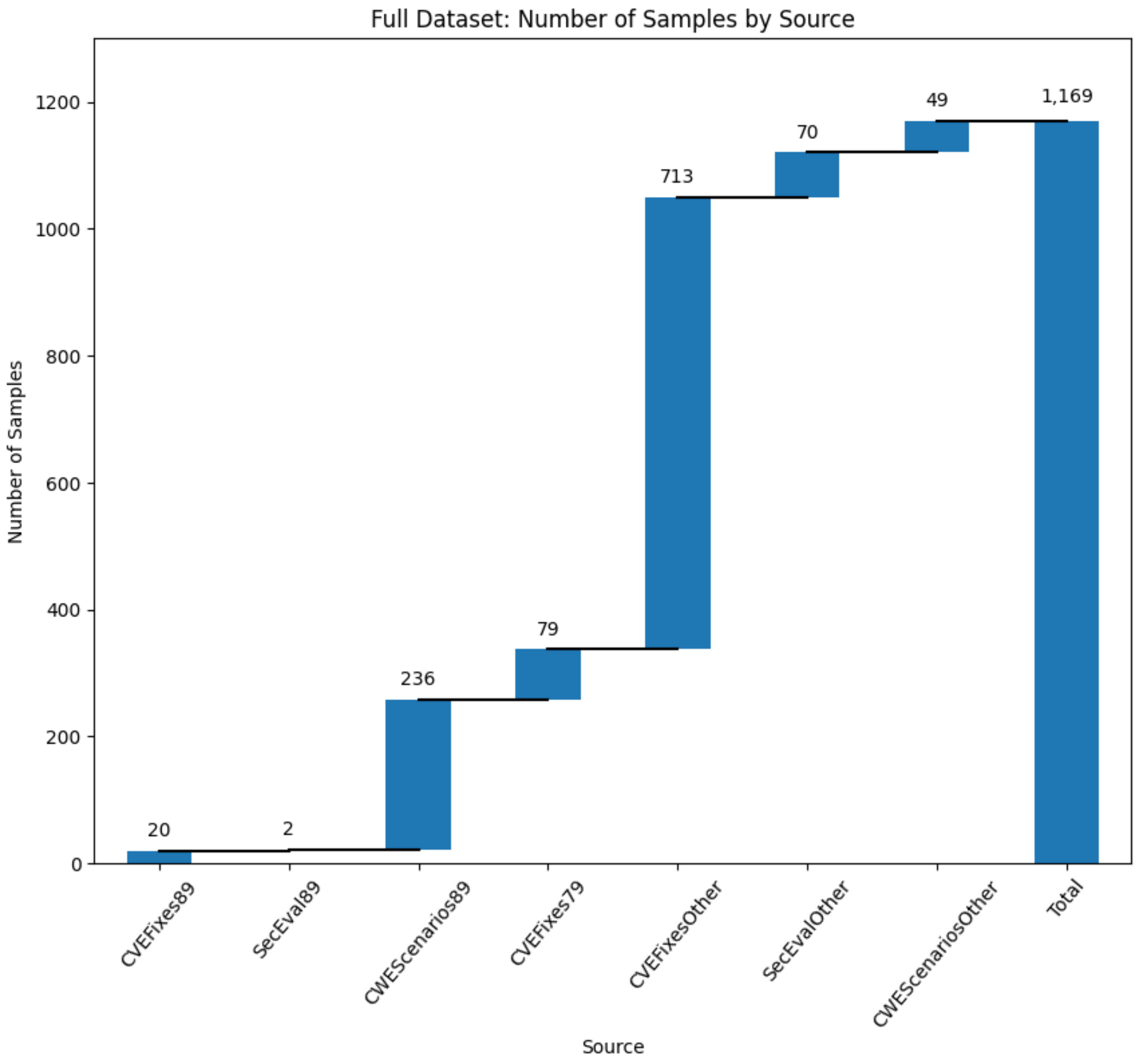

Figure 7.4: Number of Samples by Dataset and CWE

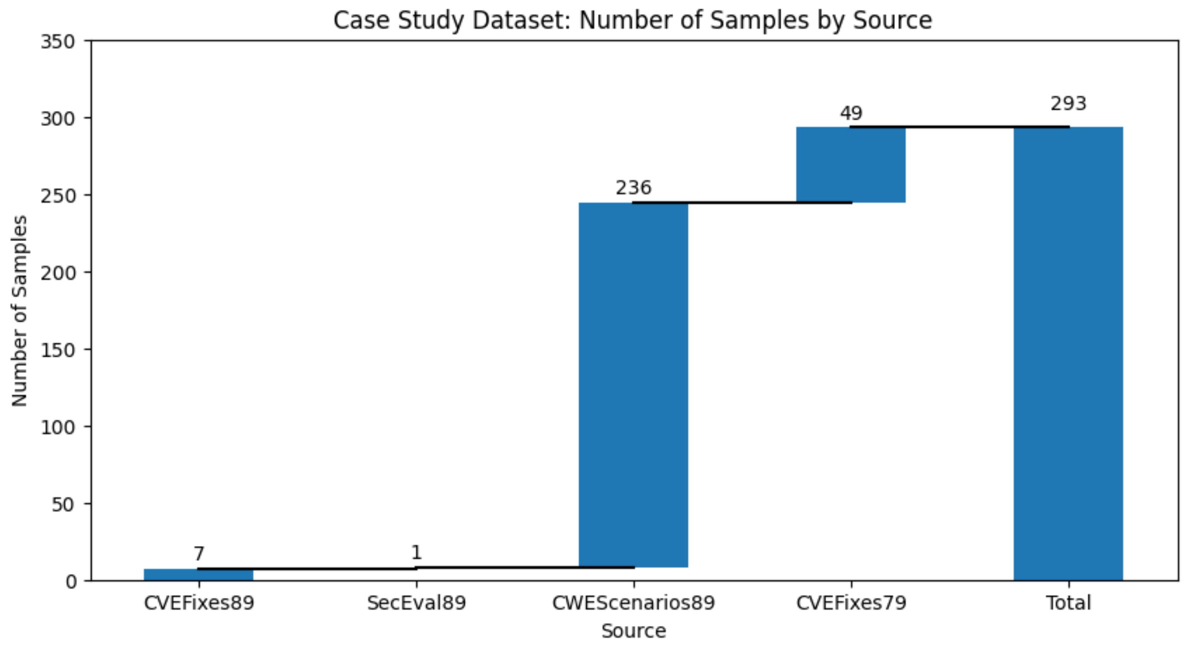

Figure 7.5: Case Study Data Code Sample Distribution

Chapter 8

Results

We first created a paired box plot to investigate rationale probability distributions by code taxonomy and CWE (Fig. 8.1). As detailed in the previous section, we applied bootstrapping with 10,000 resamples to individual CWE taxonomies; the text represents 97% and 2.5% bootstrapping confidence intervals, rounded to four decimal places. From these initial results we conducted further analysis on the 3 taxonomies for each CWE that had the highest values. These were return, asserts, and exceptions for CWE-79 and exceptions, types, and conditionals for CWE-89. Lastly, we investigated the top rationales for each CWE via bar charts. These are Fig. 8.2 for CWE-79, Fig. 8.3 for CWE-89, and Fig. 8.4 for both. The y axis is count; the individual CWE plots are color coded by taxonomy, and the overall plot is color coded by CWE.

8.1 Global Analysis

By first glance of Fig. 8.1, we can immediately identify that the syntax taxonomies of CWE-79 and CWE-89 rationales are different. Furthermore, the rationales did not map to every syntax taxonomy; there are 25 syntax taxonomies in total, but our rationales only mapped to 20 of those 25. This does not mean the 5 that did not appear in our rationales are totally irrelevant, however; asserts, bool, and with rationales were only detected for CWE-79 and loops rationales were only detected for CWE-89. Due to this we can conclude that different elements of the context have different contribution levels in generating different CWEs. Additionally, shared rationales often have varying degrees of contribution to generated weaknesses in terms of scale.

The three highest syntax taxonomy probabilities were exceptions, conditionals, and return for CWE-79 and conditionals, type, and string for CWE-89. In contrast, the widest range of values were conditionals, exceptions, and a tie between types and return for CWE-79, and asserts, exceptions, and conditionals for CWE-89. Lastly, conditionals has the most contribution to CWE-89 weaknesses, and exceptions has the most contribution to CWE-79 weaknesses.

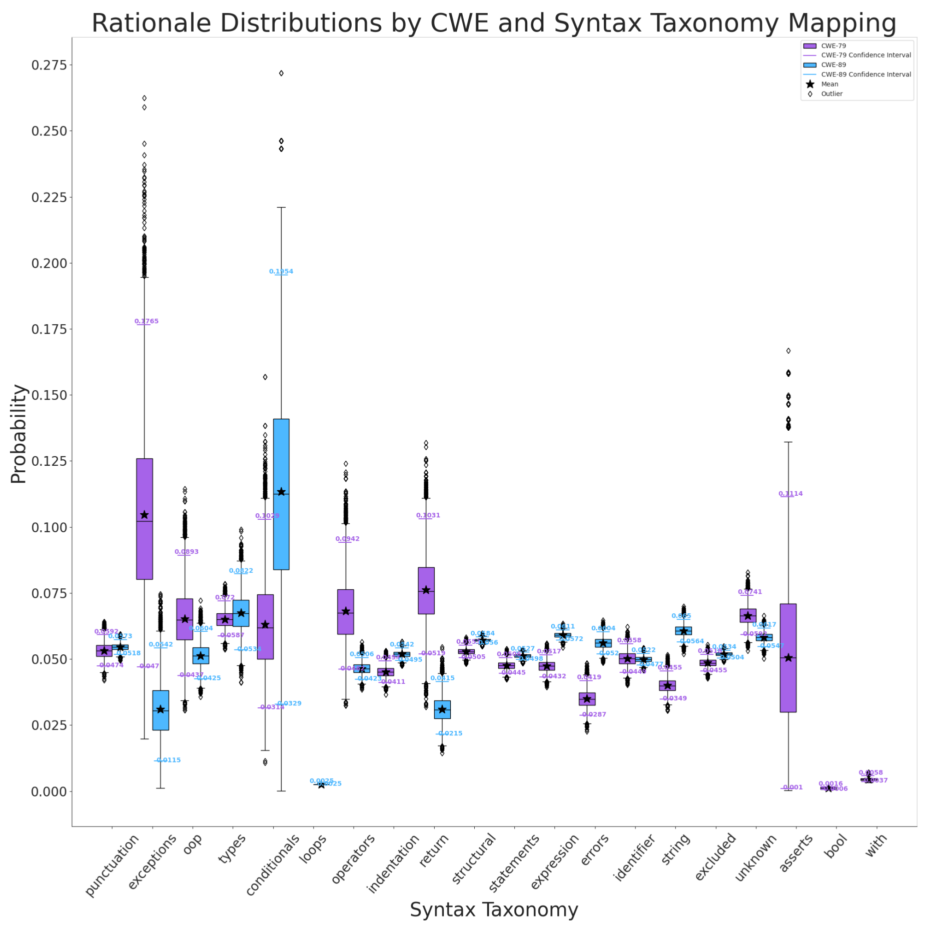

Figure 8.1: Case Study Results: Probability Distributions by Taxonomy and CWE

8.2 Local Analysis

Let us compare Fig. 8.2 and 8.3. Notice that the return rationale by far occurs the most for CWE-79, and the count drops off sharply for other rationales.[1] As initially observed during global analysis, we can see the taxonomy frequency is clearly different. Interestingly, exceptions appears twice: for CWE-79, it has the most unique rationales, and for CWE-89 it has the highest rationale count. The shared 'exceptions' rationale is 'try': thus try statements can lead to either insecurity, albeit with different commonality.

We can also compare the top rationales for CWE-79 and CWE-89 by inves-

[1]Whitespace matters - see the two entries for '.' in CWE-79 and two entries for 'try' in CWE-89.

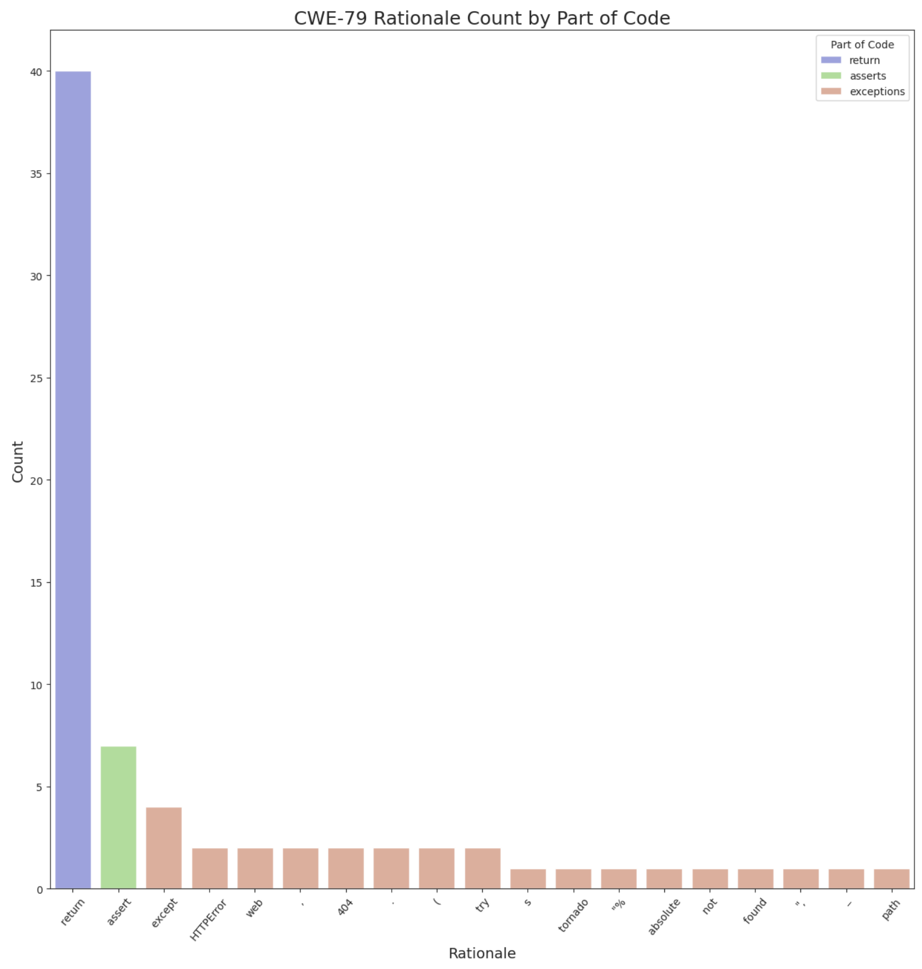

Figure 8.2: Case Study Results: Top CWE-79 Rationales

tigating Fig. 8.4. First, return does not even appear as a rationale for CWE-89 and it is by far the most common rationale for either weaknesses. This further underscores our previously discussed findings. Second, CWE-89 has a more gradual decrease in rationale count than CWE-79, which indicates SQL injection can be created due to a wide combination of rationales whereas XSS injection is more often related to the returned object. Inherently, the specific rationales make sense for the given weaknesses: SQL injection exclusive rationales such as POST and NAME are clearly from SQL statements, while XSS exclusive rationales such as HTTPError, path, and web are clearly related to web processes.

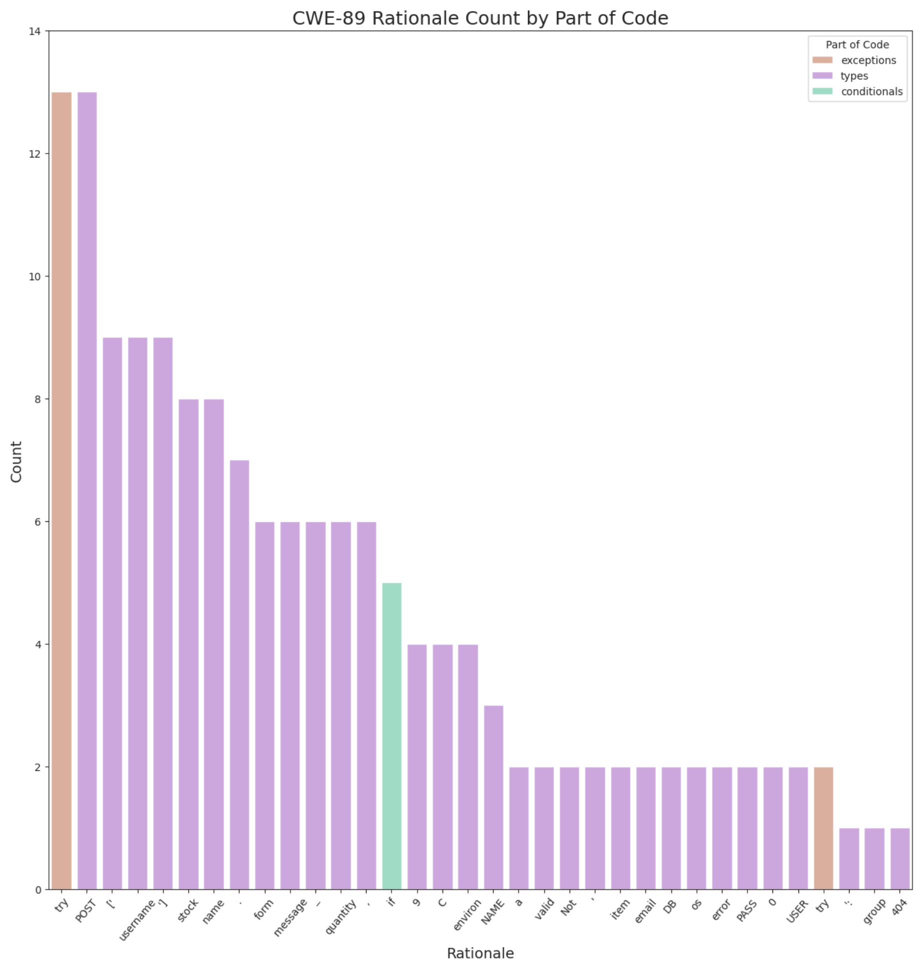

Figure 8.3: Case Study Results: Top CWE-89 Rationales

8.3 Implications

Our global and local analyses both conclude that individual weaknesses have unique rationale syntax taxonomies. This implies that there is no one-size-fits-all approach to preventing insecure code generation; instead, preventing individual weaknesses will have to occur through specialized methods. There are almost one thousand weaknesses; if they all have rationales as different as these two, identifying rationales for each one will be time-consuming and tedious. However, it could provide exponential returns on mitigating downstream security problems through targeted identification of potentially problematic tokens.

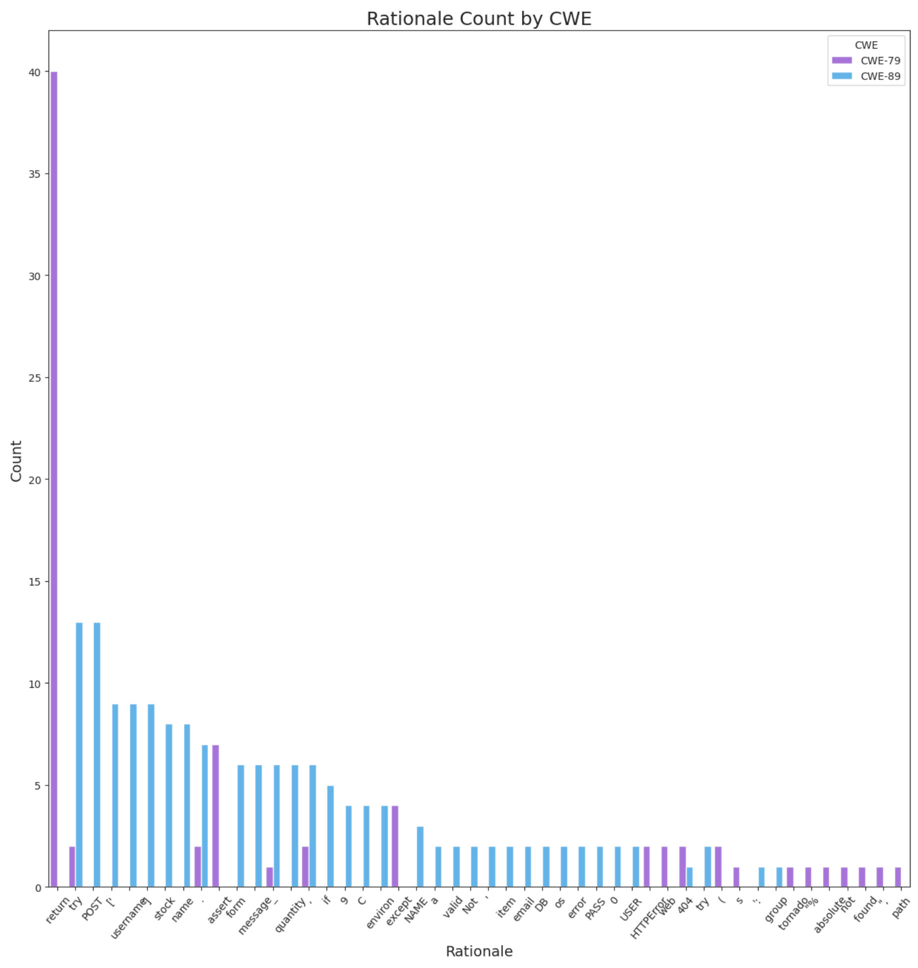

Figure 8.4: Case Study Results: Top Rationales by CWE

Chapter 9

Conclusion

Our study investigated LLM-generated code samples though applying a new interpretability method, sequential rationales. These code samples contained weaknesses, a cybersecurity term referring to code which could be exploited by a malicious actor if released as part of a product. We ran a case study of CWE-79 and CWE-89 using GPT-2 model Codeparrot on code samples from a subsection of data synthesized for the purposes of this study. The rationales for the two weaknesses have drastic differences in code taxonomies, indicating individual weaknesses arise from different tokens in the code context. More specifically, we observed that CWE-79 rationales were mostly exceptions or conditionals, while CWE-89 rationales were overwhelmingly related to conditionals.

There are many directions further research can take. The first priority should be creating more data under a centralized system of organization. There are nearly one thousand CWEs, all with incredibly diverse security concerns - from the results of our case study, it is evident that each one might have different patterns in their rationales. Without more data we cannot know. A subsection of this is developing novel vulnerability span extraction tools, as manual review is time-consuming and repetitive. The second priority is connected to the first, and it is to extend analysis to 1) more CWEs, and 2) determining the most useful type of aggregation functions. Third, creating a framework for developers to flag when the code context could lead to LLM-generated weaknesses would be immensely useful in the years to come as Copilot and other AI assistant developers become more popular. Lastly, our initial evaluation only used one model; thus, further research should extending analysis to multiple models and thereby generalize results.

Appendix A: Source Dataset Boxplots

We would be remiss to not mention several observations. First, there are zero AST errors in the CVEFixes dataset for CWE-79, even with bootstrapping. This peculiarity is likely due to the nature of CVEFixes' data gathering format in contrast to our other two sources - detection of real vulnerabilities instead of examples of weaknesses generated for study and experimentation. As such, it is unlikely deployed code has compilation errors. Second, the combined CWEScenarios/SecEval dataset does not contain any CWE-79 samples. As such, it mostly consists of CWEScenarios data.

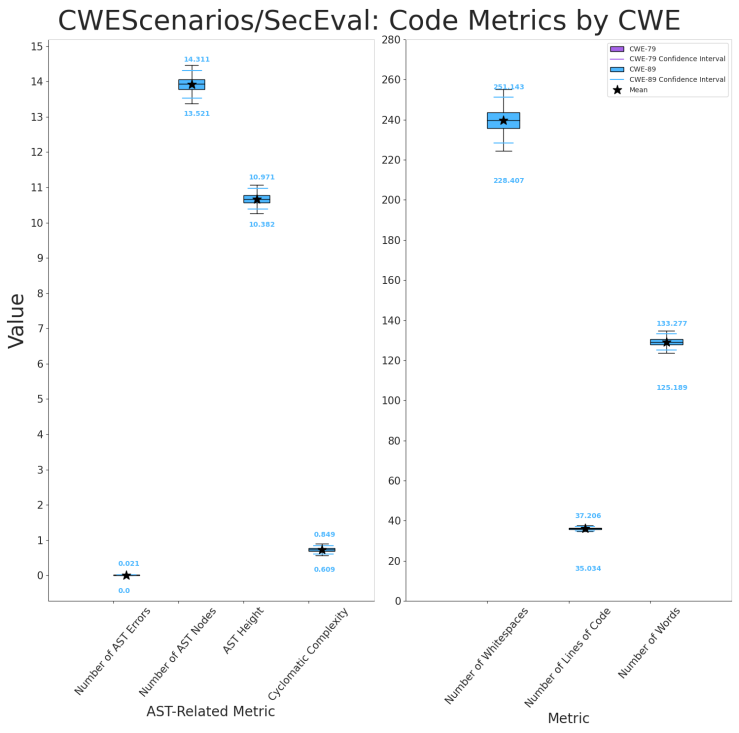

Figure 9.1: CWEScenarios/SecEval Dataset: CWE-79 and CWE-89

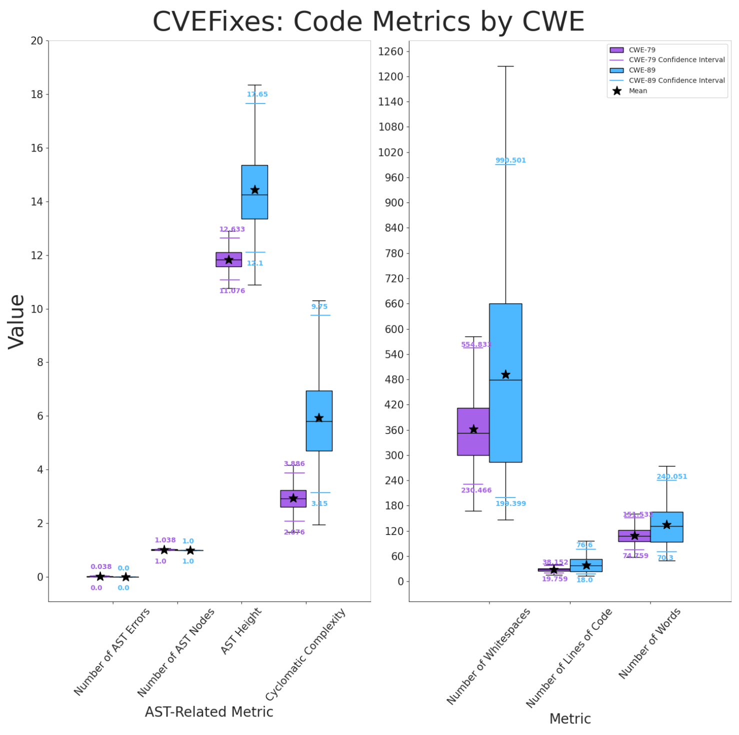

Figure 9.2: CVEFixes Dataset: CWE-79 and CWE-89

Appendix B: SOTA LLM Architectures

Transformers

Transformers are the go-to LM architecture for text generation. There are two key parts to the transformer model: its encoder/decoder blocks and the attention mechanism. The most basic Transformer architecture is illustrated in Fig. 9.4. The left side of the figure is an encoder block, and the right side is a decoder block.

Encoder Blocks First, we embed each individual entry of our input sequence (tokens) into tensors so the model can make calculations. To complete our embedding we must add some positional encoding to the resultant input vectors such that information about their original location in the sequence is preserved. Then we enter the encoder block, where we split the inputs into 3 vectors (QKV), calculate multi-head attention (MHA), then add the original input to the MHA output and normalize the result. This adding and normalization layer is residual in nature. At the end of the encoder block, we have obtained a continuous encoding of our input sequence in the model dimension.[1]

Decoder Blocks Entering the decoder, we send this continuous representation of our input sequence to the MHA (3rd block in the right half of Fig. 9.4), add and normalize, then send it through another residual feedforward layer. Exiting the decoder, we send the output through a linear layer then softmax activation function to obtain output probabilities. In the next iteration, the decoder outputs are sent to become the decoder input. They are shifted right to limit decoder input to $y_{<t}$ tokens for encoder input sequence $y_{1:t}$, which masked MHA further enforces.

The Attention Mechanism In the attention mechanism, we split our input sequence into three vectors: Query, Key, and Value. The query and key vectors

[1]https://jalammar.github.io/illustrated-transformer/

have the same dimensionality d_k, are multiplied together, scaled by their dimensionality, and then activated with SoftMax. The output of SoftMax is then multiplied by the value vector. The left side of Fig. 9.3 displays this process; MatMul stands for matrix product. Mathematically, the output of dot-product attention is

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V = \frac{e^{\frac{Q_i K_i^T}{\sqrt{d_k}}}}{\sum_{j=1}^{K} e^{\frac{Q_j K_j^T}{\sqrt{d_k}}}} V \tag{9.1}$$

This dot-product attention is able to capture the relative importance of elements in a sequence by scaling aspects of sequence inputs as detailed above. In other words, attention forces the model to pay attention to the most important input tokens.

Multi-Head Attention first begins by sending each vector through a linear layer, projecting them to different learned representations in the model dimension before calculating attention in parallel. This enables the model to look across the sequence to determine relevance. It is important that this does not result in the model "cheating" and looking at the answer, thus we may employ masking to remove the possibility of treating the desired output as an input. To keep complexity consistent with single-headed attention, we reduce the dimensions of the Q, K, and V vectors by the model dimensions divided by the number of attention heads [45].

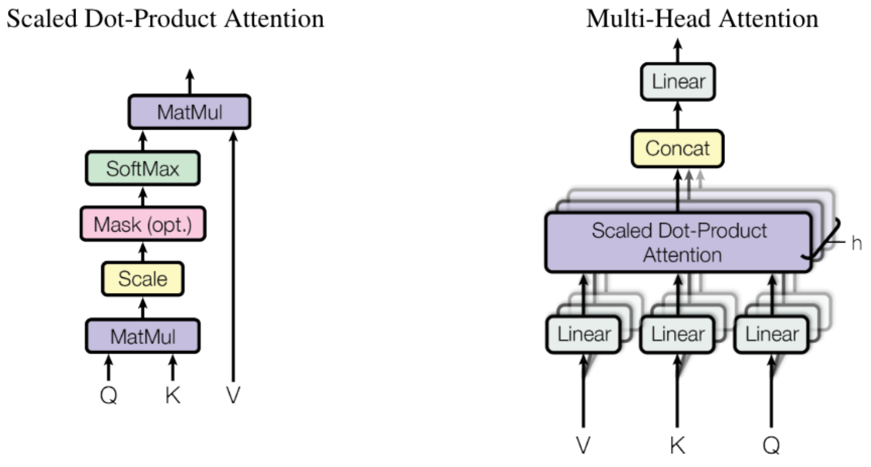

Figure 9.3: Attention (left); Multi-Head Attention in Parallel (right)

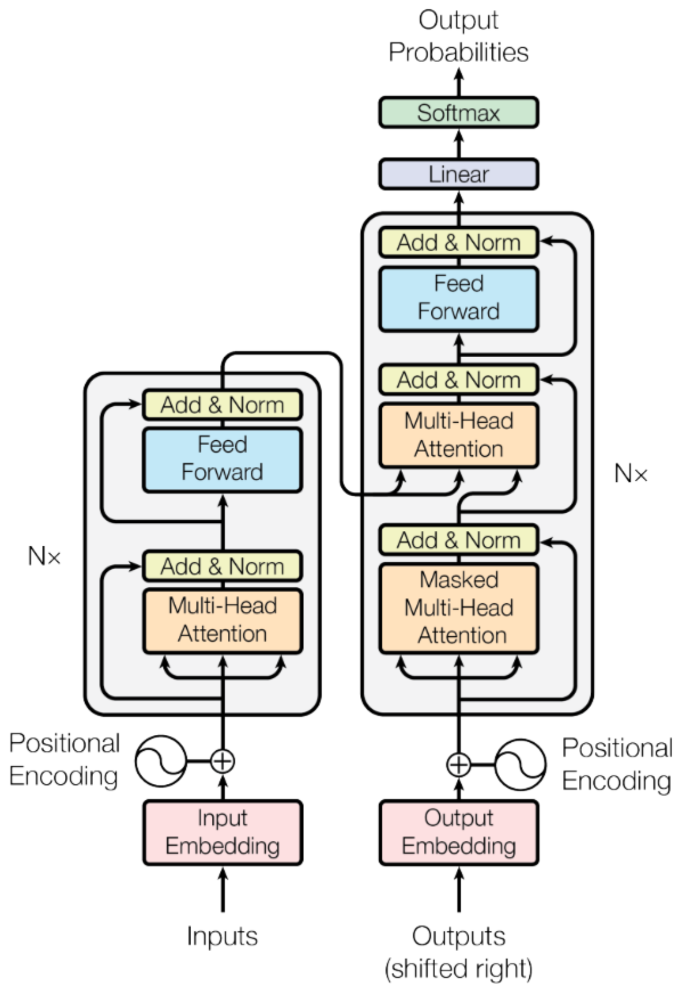

Figure 9.4: Transformer Block Architecture

GPT

ChatGPT is built on the GPT architecture, which was first introduced in February of 2019 with a full release in November of the same year. The architecture consists of twelve Transformer Decoder blocks in parallel. Its main power is its general use nature, which comes from a semi-supervised two-step training process: generative pre-training followed by discriminative fine-tuning [48].

In the first stage, the model initializes its weights through performing unsupervised learning on mass amounts of unlabeled text treated as a single, unbroken sequence of tokens. The first stage produces a LLM primed for specialization to any number of individual tasks. The second stage has everything to do with specialization of a general LLM. In this stage, the model is fine-tuned with supervised learning via use of a labeled dataset and addition of the ultimate layer: a linear layer following the final decoder activation function utilizing a Softmax activation function to maximize classification accuracy. Standard finetuning, however, will not work when faced with certain tasks such as a conversation with an end user. In these cases GPT employs input preprocessing to avoid changing the architecture itself which would be less efficient. The process of specializing an all-purpose model detailed here is called transfer learning - and finetuning is an example of it.

GPT-2

The initial success of GPT lead to more experiments with the architecture. Its successor, GPT-2, was given a larger and more diverse dataset: the text content from webpages of 45 million outgoing Reddit links with 3+ upvotes. Providing the GPT architecture with such varied inputs lead to better application to more generation tasks. GPT-2 differs from GPT in a few hyperparameters as well; most notably, the vocabulary expanded to 50,257 and the context size doubled [60]. GPT and GPT-2's abilities can be described as multitasking. Many models available on HuggingFace are based on GPT-2, as it can be finetuned to have high performance on pretty much any text-generation related task.

Bibliography

[1] F. Duarte, “Number of chatgpt users (apr 2024).” [Online]. Available: https://explodingtopics.com/blog/chatgpt-userstop

[2] OpenAI, “Introducing chatgpt.” [Online]. Available: https://openai.com/blog/chatgpt

[3] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, “A systematic literature review on the use of deep learning in software engineering research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1145/3485275

[4] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, L. Wang, A. T. Luu, W. Bi, F. Shi, and S. Shi, “Siren’s song in the ai ocean: A survey on hallucination in large language models,” 2023.

[5] GitHub, “The world’s most widely adopted ai developer tool.” [Online]. Available: https://github.com/features/copilot

[6] Microsoft, “Type less, code more.” [Online]. Available: https://visualstudio.microsoft.com/services/intellicode/

[7] Tabnine, “The ai coding assistant that you control.” [Online]. Available: https://www.tabnine.com/

[8] OpenAI, “Openai codex.” [Online]. Available: https://openai.com/index/openai-codex/

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions.” IEEE Computer Society, May 2022, pp. 754–768. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2022/131600a980/1FlQxERjKCs

[10] T. M. Corporation, “Cwe glossary.” [Online]. Available: https://cwe.mitre.org/documents/glossary/index.htmlInsecure

[11] MITRE, “Introduction to vulnerability theory.” [Online]. Available: https://cwe.mitre.org/documents/vulnerability$_{t}heory/intro.html$

[12] K. Vafa, Y. Deng, D. M. Blei, and A. M. Rush, "Rationales for Sequential Predictions," Nov. 2021, arXiv:2109.06387 [cs]. [Online]. Available: http://arxiv.org/abs/2109.06387

[13] Wikipedia, "Natural language." [Online]. Available: https://en.wikipedia.org/wiki/Natural$_l$*anguage*

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.

[15] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward Deep Learning Software Repositories," in *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR'15)*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[16] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[17] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs." [Online]. Available: http://arxiv.org/abs/1711.00740

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[20] J. He and M. Vechev, "Large Language Models for Code: Security Hardening and Adversarial Testing," Sep. 2023, arXiv:2302.05319 [cs]. [Online]. Available: http://arxiv.org/abs/2302.05319

[21] D. N. Palacio, N. Cooper, A. Rodriguez, K. Moran, and D. Poshyvanyk, "Toward a Theory of Causation for Interpreting Neural Code Models," Feb. 2023, arXiv:2302.03788 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2302.03788

[22] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What Do They Capture? – A Structural Analysis of Pre-Trained Language Models

for Source Code," Feb. 2022, arXiv:2202.06840 [cs]. [Online]. Available: http://arxiv.org/abs/2202.06840

[23] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk, "Evaluating and Explaining Large Language Models for Code Using Syntactic Structures," Aug. 2023, arXiv:2308.03873 [cs]. [Online]. Available: http://arxiv.org/abs/2308.03873

[24] J. Bastings and K. Filippova, "The elephant in the interpretability room: Why use attention as explanation when we have saliency methods?" pp. 149–155, 2020, arXiv: 2010.05607.

[25] A. H. Mohammadkhani and H. Hemmati, "Explainable AI for pre-trained code models: What do they learn? when they do not work?"

[26] A. K. Mohankumar, P. Nema, S. Narasimhan, M. M. Khapra, B. V. Srinivasan, and B. Ravindran, "Towards Transparent and Explainable Attention Models," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4206–4216. [Online]. Available: https://aclanthology.org/2020.acl-main.387

[27] S. Jain and B. C. Wallace, "Attention is not Explanation," May 2019, arXiv:1902.10186 [cs]. [Online]. Available: http://arxiv.org/abs/1902.10186

[28] S. Serrano and N. A. Smith, "Is Attention Interpretable?" Jun. 2019, arXiv:1906.03731 [cs]. [Online]. Available: http://arxiv.org/abs/1906.03731

[29] I. B. M. Corporation, "What is cybersecurity?" [Online]. Available: https://www.ibm.com/topics/cybersecurity

[30] T. M. Corporation, "Glossary." [Online]. Available: https://www.cve.org/ResourcesSupport/Glossary

[31] T. O. W. A. S. P. Foundation, "About the owasp foundation." [Online]. Available: https://owasp.org/about/

[32] ——, "Owasp top ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[33] ——, "Owasp top 10 for large language model applications." [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[34] N. P. A. Office, "Nist general information." [Online]. Available: https://www.nist.gov/director/pao/nist-general-information

[35] N. I. of Standards and Technology, "National vulnerability database." [Online]. Available: https://nvd.nist.gov/

[36] T. M. Corporation, "Cybersecurity." [Online]. Available: https://www.mitre.org/focus-areas/cybersecurity

[37] ——, "Stubborn weaknesses in the cwe top 25." [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023$_{s}tubborn_{w}eaknesses.html$

[38] ——, "Cwe category: Data neutralization issues." [Online]. Available: https://cwe.mitre.org/data/definitions/137.html

[39] ——, "Cwe view: Weaknesses in the 2023 cwe top 25 most dangerous software weaknesses." [Online]. Available: https://cwe.mitre.org/data/definitions/1425.html

[40] ——, "Cwe-79: Improper neutralization of input during web page generation ('cross-site scripting')." [Online]. Available: https://cwe.mitre.org/data/definitions/79.html

[41] Snyk, "Dom xss." [Online]. Available: https://learn.snyk.io/lesson/dom-based-xss/

[42] U. G. F. T. Commission, "Phishing." [Online]. Available: https://www.ftc.gov/business-guidance/small-businesses/cybersecurity/phishing

[43] T. M. Corporation, "Cwe-89: Improper neutralization of special elements used in an sql command ('sql injection')." [Online]. Available: https://cwe.mitre.org/data/definitions/89.html

[44] ——, "Trends in real-world cwes: 2019 to 2023." [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023$_{t}rends.html$

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[46] Y. Lecun, "A theoretical framework for back-propagation," in *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, D. Touretzky, G. Hinton, and T. Sejnowski, Eds. Morgan Kaufmann, 1988, pp. 21–28.

[47] D. Bergmann, "What is fine-tuning?" [Online]. Available: https://www.ibm.com/topics/fine-tuning

[48] A. Radford, "Improving language understanding by generative pre-training," San Francisco, California, Tech. Rep., 2018.

[49] CodeParrot, "Codeparrot." [Online]. Available: https://huggingface.co/codeparrot

[50] ——, "codeparrot." [Online]. Available: https://github.com/huggingface/transformers/tree/main/examples/research$_{p}rojects/codeparrot$

[51] ——, "codeparrot-clean." [Online]. Available: https://huggingface.co/datasets/codeparrot/codeparrot-clean

[52] U. of Washington, “Abstract syntax trees.” [Online]. Available: https://courses.cs.washington.edu/courses/cse401/08wi/lecture/AST.pdf

[53] Tree-Sitter, “Introduction.” [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[54] Radon, “Introduction to code metrics.” [Online]. Available: https://radon.readthedocs.io/en/latest/intro.htmlcyclomatic-complexity

[55] M. L. Siddiq and J. C. S. Santos, “Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques,” in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, ser. MSR4PS ’22. New York, NY, USA: ACM, 2022. [Online]. Available: https://doi.org/10.1145/3549035.3561184

[56] GitHub, “Codeql documentation.” [Online]. Available: https://codeql.github.com/docs/

[57] SonarSource, “Sonarsource static code analysis.” [Online]. Available: https://rules.sonarsource.com/

[58] G. Bhandari, A. Naseer, and L. Moonen, “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software,” in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE ’21)*. ACM, 2021, p. 10.

[59] S. Lomuscio, “Bootstrap estimates of confidence intervals.” [Online]. Available: https://library.virginia.edu/data/articles/bootstrap-estimates-of-confidence-intervals

[60] A. Radford, “Language models are multitask learners,” San Francisco, California, Tech. Rep., 2018.