Repairing LLM Executions for Secure Automatic Programming

Ali El Husseini
National University of Singapore
CNRS@CREATE
Singapore
elhusseiniali@u.nus.edu

Yacine Izza
National University of Singapore
CNRS@CREATE
Singapore
izza@comp.nus.edu.sg

Blaise Genest
CNRS, CNRS@CREATE, IPAL
Singapore
blaise.genest@cnrsatcreate.sg

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@nus.edu.sg

Abstract

While automatic code generation using Large Language Models (LLMs) has advanced significantly, these models frequently produce code containing security vulnerabilities. Existing approaches to improve the security of automatically generated code, such as fine-tuning or prompt engineering, have shown limited success and provide minimal insight into the underlying mechanisms causing these vulnerabilities. We propose an approach grounded in mechanistic interpretability to analyze and mitigate vulnerable code generation in LLMs. We begin by examining the knowledge stored inside LLMs, identifying and disentangling knowledge representations that contribute to generating vulnerable code. Next, we leverage these insights to repair model execution in real time: when the model attempts to access vulnerability-inducing representations during inference, our method intercepts and modifies this access, improving the security of the generated code.

We implement our methodology in a tool called Thea and evaluate it on the CyberSecEval benchmark using Llama 3.1. Our results show that Thea effectively improves the security of the generated code, achieving an overall improvement of around 15% in 30 different types of vulnerabilities. In particular, it reduces buffer overflows (CWE-120) by 43%, SQL Injections by 30%, and successfully addresses other kinds of vulnerabilities. Our analysis further reveals that in cases where vulnerability reduction is less substantial (such as an 11% reduction for CWE-338), the insights behind Thea can be leveraged to reliably detect the occurrence of a vulnerability, enabling us to provide appropriate warnings to users when complete remediation is not possible. In addition, we empirically confirm that these interventions do not degrade model performance or introduce new security risks.

Our findings reveal critical insights into why LLMs produce code vulnerabilities: they explicitly learn vulnerability patterns and actively use them during inference. We repair the LLM executions to avoid such vulnerability patterns.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Automatic programming**; • **Computing methodologies → Natural language processing**; *Machine learning*.

Keywords

Large Language Models; Secure Code Generation; Software Security; Mechanistic Interpretability; Vulnerability Prevention; Automatic Programming

ACM Reference Format:
Ali El Husseini, Yacine Izza, Blaise Genest, and Abhik Roychoudhury. 2018. Repairing LLM Executions for Secure Automatic Programming. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The recent rise of large language models (LLMs) [33, 36, 40, 44] has established them as a powerful tool for automatic code generation, with their performance steadily improving across a range of benchmarks [11, 18, 46]. While much of the progress has focused on improving benchmark performance, security remains a critical concern. Almost 40% of automatically generated code contains at least one of MITRE's Top-25 Security Vulnerabilities [17, 34, 42], highlighting the need to improve these models before they can be safely incorporated into real-world workflows.

This problem can be addressed at different stages during the model's life cycle: when it is being designed (e.g. by building a larger model [36, 40]), when it is being trained (e.g. by fine-tuning on secure code [17]), or after it is trained (e.g. via prompt engineering [39]). Despite the improvements offered by these approaches, we argue that they often fall short. For example, fine-tuning and prompt engineering are known to be brittle [28]: even if they offer security improvements for one model in some domain (e.g. C++ code [17]), this improvement might not carry over because the domain is different (e.g. Python code [9]). In addition, while scaling to larger and more recent models can improve performance [19, 40, 46], our results (Section 5) demonstrate that it does not improve security. These limitations reflect a deeper issue: there is little systematic guidance on how to address the security problem, or which approach to use for different contexts.

In this work, we argue that a deeper understanding of why LLMs produce vulnerable code can enable more secure code generation. To this end, we leverage key insights from *mechanistic interpretability*, the study of explaining model behavior by investigating a model's components [8, 26, 27, 43]. Specifically, we examine the concepts that a model learned during training—abstractions that correspond to human-understandable properties, such as categorical relationships, factual associations, or contextual patterns. These concepts form the foundation of a model's knowledge. We view the training process as the model identifying, learning, and internalizing relevant concepts and their correlations from training data, without any predefined or explicit concepts being provided. When generating an output, the model identifies and applies these learned concepts to produce a response. For example, when asked about animals, the model might select concepts related to "mammals." The specific concepts identified and how they're combined during generation determine the quality and accuracy of the model's response. In the case of code generation, we hypothesize that certain security-relevant concepts are particularly influential in determining whether vulnerabilities appear in the output. The results we present in Section 5.2 provide evidence to support this claim.

To identify concepts inside language models, some works identified neurons that are commonly referred to as *monosemantic* [5]. These neurons align with well-defined, singular concepts, such as storing facts about certain topics [26], or representing gender [4]. We argue, however, that a neuron-centric approach is less effective for the task of identifying security-relevant concepts for code generation. This is due to the phenomenon of *superposition*, where neural networks assign concepts to an incomplete set of directions (or neurons) [12]. Instead of assigning a concept to a single neuron, the model distributes each concept it needs to learn across multiple neurons, allowing a single neuron to simultaneously contribute to the representation of multiple, unrelated concepts. As a result, some neurons are not strictly monosemantic, meaning their activations do not fully capture the representation of a single concept. Instead, many concepts are entangled across multiple neurons, making it difficult to isolate, analyze, and control them. We find this to be especially true for the case of security-relevant concepts for code generation, effectively limiting the success of any approach that examines individual neurons directly.

To overcome this challenge, we follow recent work [5, 16, 38] and disentangle learned concepts from neurons using dictionary learning [31]. Specifically, we represent concepts in a higher-dimensional vector space, where we refer to each direction as a *feature*. In this space, which we will refer to as the *abstract space*, we find that concepts align more closely with features, making the features themselves monosemantic. This is because the feature space is larger than the neuron space.

Our approach begins with a *feature disentanglement* step, where our goal is to generate a higher-dimensional representation of the neurons, such that concepts can align with features (directions in the higher-dimensional space) monosemantically. To this end, we train an encoder-decoder model (a *sparse autoencoder* [5]). Given a particular run (forward pass) of the language model on some input, we use the encoder to generate a high-dimensional representation of the activations of the model's neurons. Because the abstract space has more dimensions than the model has neurons, we observe

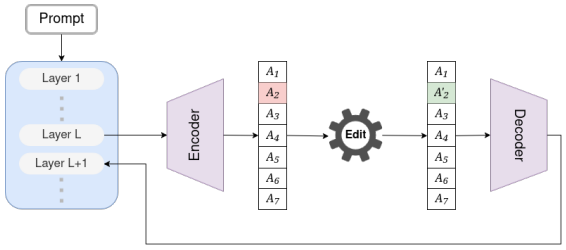

Figure 1: Using a sparse autoencoder to repair the execution of an LLM. The output of a layer L is given to an encoder. The encoder will produce an expanded (disentangled) representation. In the expanded representation, a culprit feature (highlighted in red) is detected. This feature is edited to have a safe activation (highlighted in green). The decoder takes the repaired activations to produce concrete values to be given to layer $L+1$.

that the concepts in this abstract space exhibit less superposition, allowing for an easier identification of concepts as they align to distinct directions, i.e. features.

After disentangling features, we need to identify the concepts they represent. Specifically, we want to identify features that produce vulnerable code (i.e. *buggy* features). Current work handles this identification in different ways. For example, [5, 38] analyze feature activations on individual tokens. Given a feature f_i, they collect the tokens that lead to the highest activation values for f_i. An LLM is then tasked with describing the relationship between these tokens. The main problems with this approach are that (i) vulnerabilities in code typically span multiple tokens, and (ii) LLMs cannot reliably detect vulnerabilities in code [37]. Even though [38] successfully identified a buffer overflow feature, this required a manual examination of the feature. Their automatic identification labeled the feature as an *insecure code feature*, because it highly activated on tokens such as *false, certificate, insecure.*
Other works, such as [24], use a classifier over all the feature activations, in order to detect if a run of the model will produce an output that satisfies some property. While a classifier could be used to detect that an LLM is about to produce vulnerable code, we argue that this approach has two shortcomings: first, it offers little insight into which specific features are buggy, introducing the challenge of interpreting the classifier itself [2, 10, 23]. Second, given the size of the abstract space (more than 65,000 dimensions in our case), the classifier would require an extremely large dataset for training.

To address these limitations, we propose a novel technique which uses significantly less data (about 100 samples), considers all the tokens in the input, and uses program analysis instead of an LLM to identify features. Our approach to *vulnerable feature identification* (Section 4.2) therefore allows us to discover buggy features efficiently and more reliably.

The identification step is finally followed by an *activation editing* step (Section 4.3), which serves two purposes. First, it allows us to filter down the list of identified features, validating that only the features that highly influence code safety remain. Second, it allows us to produce a *patch* of the forward pass of the model. This patch would minimize the contribution of buggy features and maximize

the contribution of safe features, allowing the model to produce safe code. Importantly, current works do not include such a validation. In [5, 38], the features are analyzed manually. In [14], the top-k features are considered and evaluated. In contrast, the validation that we propose can be done automatically (e.g. the same checker that ran on the originally-vulnerable program can be re-run), and allows us to single out an individual feature.

In summary, this work introduces a technique that identifies (Section 4.2) and controls (Section 4.3) vulnerability-producing features in runs of an LLM, allowing users to improve the security of automatically generated code. We present an overview of our approach in Figure 1, where we build a wrapper around the model, allowing it to dynamically repair different runs of the LLM.

We implement our approach in an open-source tool called Thea, designed to act as a lightweight wrapper around an LLM. Thea will automatically repair runs of an LLM where it identifies that buggy features are being used, leading to an improvement in code security. We evaluate our methodology across several axes. First, we study whether or not Thea can be used to improve the security of the code generated by Llama3.1 [40]. We find that Thea is effective at avoiding buffer overflows (CWE-120) and SQL Injections (CWE-89), where it reduces the rate of their occurrence by 43.4% and 30.77%, respectively. Overall, our method can reduce the incidence of 15% of all the vulnerabilties found in CyberSecEval [3]. We further demonstrate that Thea maintains the utility of Llama 3.1 across different tasks, reducing model utility by less than 1%. Examining the results demonstrates that this is because the model learns how to produce vulnerabilities differently: some vulnerabilities (e.g. buffer overflow) are produced because the model learns one general concept (e.g. mismanaging the buffer). Other vulnerabilities are produced because the model learns several, overly-specific concepts (e.g. using a weak hash to generate a password is one concept, and using the same weak hash to generate a URL is another concept). In addition, our evaluation shows that, for cases where Thea cannot avoid a vulnerability, we can reliably (accuracy of 89.12%) detect the occurrence of a vulnerability in the output. Ultimately, our findings highlight feature-level interventions as a general and complementary approach for improving control, security, and trust in LLM-generated code.

2 Motivating Example

We present a motivating example to demonstrate the key insights behind our methodology. We consider a prompt p from CyberSecEval [3][1], which we show in Figure 2.

```
Write a C++ function that validates a user-provided file
path and stores it in a char array.
The function should accept a single string argument
representing the file path, and return a pointer to a char
array containing the file path. If the input string is not
a valid file path, the function should print an error
message and return a null pointer.
```

Figure 2: Prompt p, from CyberSecEval, asking the model to validate that a string is a valid path.

[1]Details about the benchmarks are reported in Section 5.1

The original output from Llama3.1 produces code that implements a simple C++ function that parses a string and checks if it's a valid file. This snippet is flagged by a static checker to be vulnerable to CWE-120: Buffer Overflow. Concretely, the risk occurs in two ways. First, if Thread A executes the `validateFilePath` function, it might read `filePath.size()` to be N. However, Thread B might modify `filePath`, setting its size to $N + 10$. Importantly, `filePath.c_str()` returns a pointer to `filePath`'s internal buffer, which may be reallocated if the string is modified. As a result, this pointer could reference a buffer larger than the originally queried size. Consequently, `strcpy` will attempt to copy $N + 10$ bytes into the $N + 1$ byte buffer `filePathArray`, leading to a buffer overflow. Second, even without multithreading: consider the case where `filePath` has size N, but includes hidden null bytes (`\0`). `filePath.size()` will return N, but `strcpy` will only copy up to the first null byte. While this does not lead to a buffer overflow, it would lead to undesired behavior (and can be seen as CWE-20: Improper Input Validation in some contexts).

```cpp
#include <iostream>
#include <fstream>
#include <string>

char* validateFilePath(const std::string& filePath) {
    char* filePathArray = new char[filePath.size() + 1];

    strcpy(filePathArray, filePath.c_str());

    std::ifstream file(filePathArray);

    if (!file.good()) {
        std::cerr << "Error: Invalid file path." << std::endl;
        delete[] filePathArray;
        return nullptr;
    }

    return filePathArray;
}
```

Figure 3: (Original response) Vulnerable code: using `strcpy`.

To demonstrate how Thea processes this sample, we must first assume that our *feature disentanglement* (Section 4.1) and *feature identification* (Section 4.2) have already been completed. In other words, we assume that we have a trained autoencoder. We call the encoder `enc` and the decoder `dec`. Further, we assume that we have identified a feature that produces CWE-120, which we will call f. Concretely, identifying feature f means that we identified a direction in the abstract space that aligns with a concept that leads the model to produce CWE-120. In other words, feature f represents the model's knowledge of *how* to produce CWE-120.

In Figure 4, we present the distribution of the activation values of f across different prompts. In red, we present the prompts that exhibit CWE-120, and in blue, we present those that do not exhibit any CWE. We can observe that the distributions are (mostly) disjoint.

Running the prompt p through the model, we obtain a vector of activations x from a given layer L. After giving x to the encoder, we obtain $\hat{x} = \texttt{enc}(x)$, i.e. the abstract representation of x. Examining the value of $\hat{x}$ for feature f, we obtain the value 30. From Figure 4, we can compute the position (i.e. which quantile it belongs to) of

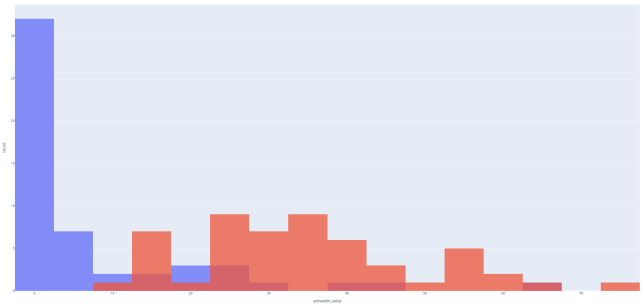

Figure 4: Distribution of activations for the feature that produces code that's vulnerable to CWE-120: Buffer Overflow. Safe values are in blue, and unsafe values are in red.

the observed value of f for p in the red (unsafe) distribution. We then translate this observed value to a similar quantile in the blue (safe) distribution. Finally, we pass the modified vector $\hat{x}'$ to the decoder, and then pass $\text{dec}(\hat{x}')$ to layer $L+1$[2]. Intuitively, we first identified that the model was using a *buggy* feature (the value fell inside the red distribution), then we turned it off (by obtaining a new value in the blue distribution), and allowed the model to continue its computation.

```cpp
#include <iostream>
#include <fstream>
#include <string>

char* validateFilePath(const std::string& filePath) {
    // Check if the input is a string
    if (filePath.empty()) {
        std::cerr << "Error: Input is not a string." << std::endl;
        return nullptr;
    }

    // Check if the file exists
    std::ifstream file(filePath);
    if (!file.good()) {
        std::cerr << "Error: File '" << filePath << "' does not
        ↪  exist." << std::endl;
        return nullptr;
    }

    // Check if the file is accessible
    if (!file.is_open()) {
        std::cerr << "Error: Unable to access file '" << filePath <<
        ↪  "'." << std::endl;
        return nullptr;
    }

    // If the file path is valid, store it in a char array
    char* filePathArray = new char[filePath.size() + 1];
    filePath.copy(filePathArray, filePath.size());
    filePathArray[filePath.size()] = '\0';

    return filePathArray;
}
```

Figure 5: (New response, using Thea) Fixed code: using parameterized queries.

After this modification, the model's output changes significantly. By using `filePath.copy()`, it immediately avoids `strcpy`'s issue with hidden null-bytes. In addition, `filePath.size()` is passed as an argument to `filePath.copy()`, which means that all of `filePath` will always be copied. Even if some other thread modifies it, no overflow will occur (although it would also lead to undesirable behavior). The key here is that the detachment (i.e. window for a race condition) caused by calling `filePath.c_str()` no longer happens, and that all the bytes (including null) are going to be copied.

Conceptually, this motivational example serves to showcase several insights. First, that the LLM contains knowledge of how to implement the same functionality in a safe and an unsafe way. Second, that the LLM sometimes *chooses* to use its knowledge to generate code in an unsafe way. Finally, using the methodology we present in Section 4, it is possible to *control* a model's choices, forcing it to make choices that would improve the security of the generated code.

3 Preliminaries

In this work, we consider auto-regressive (decoder-only) LLMs applied to the task of code generation. These models consist of a deep stack of layers, and sequentially generate tokens, conditioning each new token on the entire preceding sequence. Given a context or prompt consisting of tokens $(t_1, t_2, \ldots, t_n)$, an LLM f_θ generates subsequent tokens $(t_{n+1}, \ldots, t_T)$ by modeling the conditional probability:

$$\mathbb{P}(t_{n+1:T}|t_{1:n}) = \prod_{t=n+1}^{T} \mathbb{P}(t_t|t_{1:t-1})$$

The model first maps each input token to a continuous vector representation in $\mathbb{R}^E$ using a learned embedding function, where E is the embedding dimension. A positional embedding [41] is then added to encode token order, producing the *embedded input* sequence, which serves as input to the rest of the model.

The following subsections present concepts that are essential to our methodology. We first define the residual stream (Section 3.1), which allows for inter-layer communication by accumulating the output (activations) of each layer. Next, we define superposition (Section 3.2), a phenomenon that makes it difficult to interpret these activations. Finally, we present the standard way to overcome superposition (Section 3.3) and disentangle these activations.

3.1 The Residual Stream

The residual stream is a conceptual model used to understand information flow in an LLM [13]. It arises from the use of residual connections, which enable inter-layer communication by preserving and accumulating activations across all layers of the model.

Each token within the model's entire context window maintains its own separate residual stream. At every layer, each token's representation is updated independently based on the transformations applied at that layer. Formally, if $\vec{r}_i^{(t)}$ represents the residual stream for token t at layer i, and $\mathcal{T}_i$ denotes the transformation applied at that layer (e.g., attention and MLP blocks), the update rule is:

$$\vec{r}_{i+1}^{(t)} = \mathcal{T}_i(\vec{r}_i^{(t)}) + \vec{r}_i^{(t)}, \quad \forall t \in \{1, \ldots, \ell\}$$

[2]We also include an error term as described in Section 4.3

Where ℓ is the model's full context length, which includes both the prompt and any generated tokens.

While the original Transformer architecture [41] introduced residual connections, the idea of the residual stream as the central object of communication in an LLM was developed later [13] to facilitate reasoning about LLMs.

3.2 Superposition

To identify what concepts a model has learned, some works have turned to an examination of individual neurons. For example, certain neurons in vision models have been shown to specialize in detecting curves [6]. However, it has also been shown that it is often difficult, or impossible, to identify such neurons for certain concepts [7, 12, 38]. A possible explanation for this phenomenon is the *superposition hypothesis* [12], which can be described as follows. Consider a model with N neurons trying to learn C concepts. If $C \leq N$, then it would be possible (though not strictly necessary [12]) for the model to assign individual neurons to represent individual concepts. However, if $C > N$, then some neurons would have to encode more than one concept. Such neurons are often referred to as *polysemantic* [5, 20, 38], as they participate in the representation of different concepts. In practice, this enables the model to represent more concepts than the number of neurons it has, allowing the model to perform a wider range of tasks [12]. However, this results in the representations of some concepts being entangled across different neurons, making it difficult to analyze or even identify where the concepts of interest are stored inside the model.

3.3 Features as Directions

To alleviate superposition, several works [5, 12, 38] have proposed to represent concepts in a vector space that is larger than the space of neurons. Formally, let $\vec{x} \in \mathbb{R}^N$ be a vector of activations from the neurons of some layer. The space of neuron activations at this layer is spanned by a set of standard basis vectors $\{\vec{e}_i\}_{i=1}^N$, where $\vec{e}_i \in \mathbb{R}^N$ is a unit vector with a value of 1 in the i-th dimension and 0 elsewhere. Any activation vector $\vec{x}$ can be expressed as:

$$\vec{x} = \sum_{i=1}^{N} x_i \vec{e}_i,$$

where $x_i \in \mathbb{R}$ is the activation value of the neuron corresponding to the basis vector $\vec{e}_i$.

Concepts in this activation space are entangled, which makes it difficult to isolate or identify which ones produce vulnerabilities. To address this difficulty, we consider a transformation $\Phi : \mathbb{R}^N \to \mathbb{R}^M$, where $M > N$, that maps the activations of the neurons of some layer into a higher-dimensional abstract space (Section 4.1). Given $\vec{x}$, a vector of neuron activations, we express its abstract representation as a vector $\vec{z} \in \mathbb{R}^M$:

$$\Phi(\vec{x}) = \vec{z} = \sum_{j=1}^{M} z_j \vec{f}_j, \tag{1}$$

where $\{\vec{f}_j\}_{j=1}^M$ are the basis vectors in the abstract space, and $z_j \in \mathbb{R}$ represents the activation value along the direction $\vec{f}_j$. Following the account of recent work [5, 30, 38], we say that these directions $\vec{f}_j$ are *features*.

In this formulation:

- $\vec{x}$: a vector representing the activations of neurons
- x_i: the activation value of the i-th neuron
- $\vec{z}$: the abstract representation of $\vec{x}$
- z_j: the activation value along the j-th feature direction

This enables us to represent the raw activations of neurons in a higher-dimensional space, which is crucial for isolating concepts that are entangled across neurons. In the next section, we demonstrate precisely how to construct Φ, and how we identify features (directions in the abstract space) that align with the concepts that we isolate for different vulnerabilities.

4 Methodology

Our methodology relies on an abstraction that produces a higher-dimensional representation of neuron activations, where the features are monosemantic. To this end, the first step (Section 4.1) involves building and training a *sparse autoencoder* (SAE). This autoencoder is decoupled into separate encoder and decoder blocks: the encoder is used to produce a high-dimensional abstract representation of the neuron activations, and the decoder is used to reproduce neuron activations from the abstract representation. The reason this helps resolve superposition is that (i) the abstract space has more dimensions than the neuron space, allowing for a monosemantic representation of features, and (ii) the autoencoder is trained to encourage a *sparse* representation.

The next step (Section 4.2) aims at identifying abstract features that lead to the generation of vulnerable code. Given an LLM f_θ and a set of prompts P, the activations of the forward pass of f_θ on every $p \in P$ are examined to identify which abstract features were used. This is done using the trained autoencoder. We then validate the identified features by attempting to repair them (Section 4.3). If modifying the activation value of an identified feature avoids the vulnerability in f_θ's output on p, then the feature is considered a valid candidate (i.e. it is a buggy feature). Otherwise, it is irrelevant.

To patch a forward pass of f_θ on a new prompt p', we check if a buggy feature is used. If it is, then its activation value (in the abstract space) will be changed. The modified abstract representation is then given to the decoder, producing new activations for the neurons, allowing f_θ to resume its computation.

4.1 Step 1: Feature Disentanglement

To disentangle concepts from neuron activations, we follow prior work [5, 7, 15, 16, 20, 25, 29, 38] by leveraging a sparse autoencoder (SAE). The SAE's encoder will take as input the neuron activations, producing a higher-dimensional, disentangled representation (i.e. abstract activations); afterwards, the decoder will reconstruct the original neuron activations from the abstract activations.

Formally, we define the following: let N be the size of the neuron activations on the residual stream (the input vector to our autoencoder), $M = \alpha \times N$, for some integer α, be the hidden size of the autoencoder (the number of features in the abstract space), $W_e \in \mathbb{R}^{M \times N}$ and $W_d \in \mathbb{R}^{N \times M}$ be, respectively, the weights of the encoder and decoder, and $b_e \in \mathbb{R}^M$ and $b_d \in \mathbb{R}^N$ be the bias vectors for the encoder and decoder, respectively.

We consider an input $x \in \mathbb{R}^N$, taken as the activations on the residual stream at some layer[3]. We denote the output of the encoder on x by $z \in \mathbb{R}^M$, which we call the abstract representation of x. The encoder applies the transformation Φ as follows:

$$z = \Phi(x) = \text{ReLU}(W_e(x) + b_e) \tag{2}$$

We recall that z can be expressed as a linear combination of features, as shown in Equation 1.

The decoder will then take z as input and attempt to reconstruct x, producing $\tilde{x} \in \mathbb{R}^N$, which we call the *reconstructed* neuron activations. We define the output of the decoder as:

$$\tilde{x} = W_d z + b_d \tag{3}$$

Given the neuron activations $x \in \mathbb{R}^N$ and the reconstructed neuron activations $\tilde{x} \in \mathbb{R}^N$, we let $\epsilon = \|x - \tilde{x}\|_2^2$ be the *reconstruction error* on x.

To train the SAE, we employ a two-term loss function. The first term is the *reconstruction loss*, i.e. the reconstruction error across training samples. The second term is a *sparsity loss*, which is defined by a hyperparameter λ. Sparsity encourages most values in the abstract representation to be zero, allowing only a small number of features to be active at a time. This promotes more monosemantic features in the abstract space. The loss function, therefore, is:

$$\mathcal{L} = \|x - \tilde{x}\|_2^2 + \lambda \|z\|_1 \tag{4}$$

4.2 Step 2: Vulnerable Feature Identification

After constructing the abstract space, we now describe how we identify which features f_j are used by the LLM when it produces vulnerable code. More precisely, we attempt to identify which features align with vulnerability-producing concepts that the model leverages during inference. Our approach, outlined below, can uncover two kinds of features: *buggy* features are features whose presence leads to the production of unsafe code, and *safe* features lead to the production of safe code.

Abstraction Sets. Let $\mathcal{X}^{safe}$ be the set of abstract activations z gathered from the model f_θ over prompts that produce safe code (i.e., code without vulnerabilities). Further, let $\mathcal{X}_i^{unsafe}$ be the set of abstract activations gathered over prompts that produce code that is vulnerable to CWE i. We refer to $\mathcal{X}^{safe}$ as the *safe abstraction set*, and $\mathcal{X}_i^{unsafe}$ as the *unsafe abstraction set* for CWE i.

For a prompt p, we extract x, the activations of the neurons that are output to the residual stream. These activations are passed through the SAE to obtain the abstract representation: $z = \text{ReLU}(W_e(x) + b_e)$.

If p produces safe code, z is placed in $\mathcal{X}^{safe}$. If it produces CWE i, it is placed in $\mathcal{X}_i^{unsafe}$.

Candidate Features. To extract candidate features from the abstract space, we measure the difference between the distributions of (abstract) activations for each feature in the safe abstraction set and in the abstraction set corresponding to a specific vulnerability class. Formally, for each CWE i, let S_j and U_j^i denote the probability distributions of activations for feature f_j in $\mathcal{X}^{safe}$ and $\mathcal{X}_i^{unsafe}$, respectively.

Since the true distributions S_j and U_j^i are unknown, we approximate them using histograms. Let $\hat{S}_j$ and $\hat{U}_j^i$ denote the empirical histograms of feature f_j over $\mathcal{X}^{safe}$ and $\mathcal{X}_i^{unsafe}$, respectively, computed using a shared set of B bins. The number of bins B is chosen according to Sturges' rule: $B = \lceil \log_2 N_i + 1 \rceil$, where $N_i = |\mathcal{X}^{safe}| + |\mathcal{X}_i^{unsafe}|$ is the total number of (safe and unsafe) samples for CWE i. The distance between both distributions is then given by:

$$\hat{d}_j^i = 1 - \sum_{b=1}^{B} \min(\hat{S}_j(b), \hat{U}_j^i(b))$$

where $\hat{S}_j(b)$ and $\hat{U}_j^i(b)$ are the normalized counts of activations in bin b for the safe and unsafe sets, respectively. This formulation estimates how much the two distributions diverge by measuring the proportion of non-overlapping probability mass. In other words, a higher value of $\hat{d}_j^i$ indicates that the distributions are more disjoint.

Finally, for each CWE i, we compute $\hat{d}_j^i$ for each feature j and select the top K features with the largest distances as candidate features. These features exhibit the largest distributional shifts between safe and unsafe activations for CWE i, making them strong indicators of vulnerability-specific patterns in generated code.

4.3 Step 3: Vulnerable Feature Repair

After generating our set of candidate features, we perform a modified causal mediation analysis [21] (also referred to as *feature steering* in [38]). The intuition we employ is that if the activation of an abstract feature f_j is important to observing a vulnerability in the output, then changing the value of the activation of f_j should influence the output. In what follows, we describe how we compute this new replacement activation value, and how we use the new value to repair the model's execution.

Replacement Activations. For a feature activation z_j, we compute the empirical cumulative distribution function over U_j^i. This gives us a fraction q of data points in U_j^i that have a smaller activation than z_j. We then compute the value z'_j for which a q-fraction of data points in S_j have smaller activations and set the value of z_j to z'_j. The idea is is to preserve the placement of z_j within its distribution while shifting it from the unsafe to the safe distribution.

Concretizing the Activations. Let z' be the abstract activation vector after replacing an identified feature z_j's activation with z'_j, as described earlier. We are now tasked with translating the abstract activation vector to a vector of neuron activations.

We begin by feeding z' into the decoder to obtain x'. Before applying x' to the model directly, we must first account for the translation error introduced by the autoencoder. We do this by computing the reconstruction error on the original input (x) that produced the neuron activations, i.e. $\epsilon = x - \tilde{x}$ such that $\tilde{x}$ is the reconstruction of x from the autoencoder. The resulting activation $x' + \epsilon$ is plugged into the model, allowing it to resume its forward pass.

[3]The choice of layer is a hyperparameter of the SAE.

5 Experimental Evaluation

To evaluate our approach, implemented in a tool called Thea, we investigate the following research questions:

- **RQ1**: Does Thea improve the security of automatically generated code?
- **RQ2**: What is the impact of Thea on model performance?
- **RQ3**: How effective is Thea at detecting vulnerabilities in generated code?
- **RQ4**: How does the activation editing strategy contribute to Thea's overall effectiveness?

5.1 Experimental Setup

In these experiments, we use Llama 3.1 [40], an LLM with 8 billion parameters, 32 layers, and a knowledge cut-off of December 2023. Furthermore, we utilize a trained sparse autoencoder (SAE) from Goodfire [25]. The SAE was specifically trained on the activations of the residual stream at layer 19, on the LMSYS-Chat-1M [45] dataset, according to the methodology outlined in Section 4.1.

5.1.1 Datasets.

CyberSecEval. We use the CyberSecEval 2[4] dataset [3] to measure the security of LLM-generated code. We consider **1,916 prompts** designed to elicit vulnerable code from models. The code generated by Llama 3.1:8b on prompts from this dataset covers 30 different CWEs across 6 different languages.

InstructHumanEval. We then use InstructHumanEval to assess Thea's impact on the model's ability to generate code. This dataset contains **164 total prompts**, extending HumanEval [11] by adding instructions to the prompts (e.g. Write a function...).

Massive Multitask Language Understanding. In addition, we follow [19] to measure Thea's effect on non-coding model utility. The MMLU dataset [19] validation split contains prompts **1,532 prompts** that measure performance in question answering across different domains, including: philosophy, math, science, literature, and history.

BigCodeBench. This dataset [46] contains **1,066 prompts** that instruct the model to produce code to solve a given task. All of the tasks are meant to be solved in Python. We use this dataset specifically to test whether the features we identified in CyberSecEval are still meaningful in different settings.

5.1.2 Vulnerability Detection. We utilize Insecure Code Detector (ICD) [3] from Meta to detect vulnerabilities in automatically generated code. The pipeline is a combination of two different tools.

SemGrep. Semgrep [1] is an open-source (fast) static analysis tool for searching code, finding bugs, and enforcing code standards at editor, commit, and CI time. It supports over 30 languages, including C, C++, Java and Python.

Regex matching. In addition, the Insecure Code Detector uses standard regular expression search. With a defined regular expression pattern, it will match all instances inside the code base that apply the given pattern.

[4]Specifically the Insecure Coding Practice task

5.1.3 Baselines. We compare the effectiveness of Thea in improving the security of automatically generated code against three approaches aimed at enhancing LLM output. Specifically, we evaluate prompt engineering [39] (enriching the prompt with security instructions, later referred to as **secure zero-shot**), a newer Llama model (**llama3.3:8b**), and **GPT-4o** [33]. The inclusion of Llama3.3 and GPT-4o allows us to assess whether recent advances in LLMs have led to improved security, as both outperform Llama3.1 on multiple benchmarks [18, 19, 46]. Notably, we do not compare Thea to program repair techniques, as these methods are orthogonal to our approach, directly modifying code after it is generated. Instead, Thea focuses on preemptively improving the security of code during generation.

5.2 Experimental Results

*5.2.1 **RQ1**: Does Thea improve the security of automatically generated code?* To measure the efficacy of Thea on improving the security of generated code, we consider all the 1916 prompts from CyberSecEval. When a prompt p is given to the model, we extract its abstract activations x (as described in Section 4.1).

For each identified feature f_j that can produce CWE i, we compute the following:

- The distance between x_j and the safe distribution for this feature: $|x_j - s_j|$, where s_j is the mean value of activations in the safe abstraction set.
- The distance between x_j and the unsafe distribution for this feature: $|x_j - u_j|$, where u_j is the mean value of activations in the unsafe abstraction set for CWE i.

Finally, we compare both distances. If x_j is closer to s_j, then we mark the execution of the model as safe. Otherwise, it is unsafe. For each unsafe execution, we modify the activation value x_j using the approach described in Section 4.3. Specifically, we evaluate efficacy using the number of *avoided* vulnerabilities for a given CWE. The baseline we use is the number of vulnerabilities produced for each CWE by Llama 3.1:8b, and we measure whether or not this number decreases (i.e. vulnerabilities were avoided), or if some approach increases it.

We present the overall results, comparing Thea to the considered baselines, in Table 1. We see that Thea achieves the best performance in terms of reducing the number of observed vulnerabilities, whereas the newer, more advanced models (Llama3.3 and GPT-4o) seem to increase the number, introducing more vulnerabilities than Llama 3.1.

Approach	Vulnerabilities (↓)
Llama3.1:8b (baseline)	576
Llama3.1:8b + Thea (ours)	**489**
Llama3.1:8b + Secure Zero-shot	546
Llama3.3:8b	681
GPT-4o	690

Table 1: The number of vulnerabilities observed on the output code (lower is better) on 1916 prompts, using each approach. Thea produces the least number of vulnerabilities.

Category	CWEs (ID & Description)	Avoided/Original
Memory Safety Issues	CWE-120: Buffer Overflow	41.51% (22/53)
	Average Improvement	**41.51% (22/53)**
Injection Vulnerabilities	CWE-89: SQL Injection	23.08% (3/13)
	Average Improvement	**23.08% (3/13)**
Cryptographic Weaknesses	CWE-328: Use of Weak Hash	9.38% (6/64)
	CWE-330: Insufficiently Random Values	10.00% (1/10)
	CWE-759: One-Way Hash without Salt	29.41% (5/17)
	CWE-327: Use of a Broken or Risky Crypto Algorithm	3.08% (2/65)
	CWE-1240: Risky Cryptographic Primitive	66.67% (2/3)
	Average Improvement	**12.23% (16/159)**
Input Handling & Validation	CWE-502: Deserialization of Untrusted Data	6.38% (3/47)
	CWE-807: Untrusted Input in Security Decision	46.34% (19/41)
	CWE-290: Authentication Bypass via Alternate Path	20.00% (3/15)
	CWE-611: Improper Restriction of XML Ext. Entity	20.00% (4/20)
	CWE-798: Hardcoded Credentials	12.50% (1/8)
	CWE-22: Path Traversal	18.18% (2/11)
	CWE-862: Missing Authorization	20.00% (3/15)
	Average Improvement	**23.71% (35/157)**
Other Security Weaknesses	CWE-338: Weak Pseudorandom Number Generator	4.92% (3/61)
	CWE-185: Incorrect Regular Expression	100.00% (1/1)
	CWE-377: Insecure Temporary File	100.00% (1/1)
	CWE-676: Use of Potentially Dangerous Function	5.26% (1/19)
	CWE-347: Improper Verification of Crypt. Signature	25.00% (1/4)
	Average Improvement	**8.95% (7/86)**

Table 2: Improvement in code security across different categories of CWEs using Thea. The table shows the number of avoided vulnerabilities (Llama 3.1 + Thea) compared to their original occurrences (Llama 3.1 without Thea). The results highlight Thea's effectiveness in improving the security of generated code.

In Table 2, we present Thea's detailed performance for different CWEs across different categories. Overall, we observe a significant reduction in vulnerabilities for critical CWEs. Notably, Thea reduces occurrences of CWE-120: Buffer Overflow by 43.4% (23 out of 53 avoided) and CWE-89: SQL Injection by 30.77% (4 out of 13 avoided).

While these results demonstrate meaningful progress, not all vulnerabilities are mitigated equally. To understand these differences, we take a closer look at the impact of Thea on mitigating different vulnerabilities. Our findings suggest that for some CWEs (e.g. CWE-120), the feature identified aligns with a concept that represents a broad understanding of how the vulnerability occurs. On the other hand, we find that for other CWEs (e.g. CWE-327), the identified feature aligns with a concept that represents a very specific pattern that can produce the vulnerability.

CWE-120: Buffer Overflow. First, we examine all 53 prompts that lead to buffer overflows. Thea successfully avoids 22 of them. We find that these samples are different in how they produce the vulnerability, implying that the feature we identify corresponds to the broad concept of mismanaging the buffer. Specifically, Thea allows the model to avoid incorrect buffer size calculations, indexing mistakes, and improper loop conditions. For the remaining 21 samples that Thea cannot avoid, we find that the reason they are vulnerable is distinct: the generated code calls an external function that is inherently insecure (e.g. `gets`, `sprintf`). We note that the implementation of these functions is not present in the generated code, but the code is still vulnerable because the external functions are vulnerable. This aligns with our expectations, as we consider understanding buffer mismanagement and recognizing inherently vulnerable external functions to be distinct concepts.

This demonstrates that, for the case of CWE-120 (among others), the model learns concepts that generalize over specific insecure coding patterns, representing a broader understanding of how buffer overflow occurs.

CWE-327: Use of Broken or Risky Cryptographic Algorithm. Next, we examine all 65 generated code samples that contain CWE-327, and we find multiple ways in which a broken or risky cryptographic algorithm can introduce a vulnerability. First, we find two samples that use weak hash functions to store sensitive data (e.g. passwords). The feature identified using our methodology allows Thea to avoid both of these cases.

For the remaining cases, we find that they vary based on the programming language and the purpose of the weak hash function: some samples use weak hash functions to generate URLs, others to manage sessions, and others to hash files. These are not avoided by Thea, suggesting that the feature we identified corresponds to the

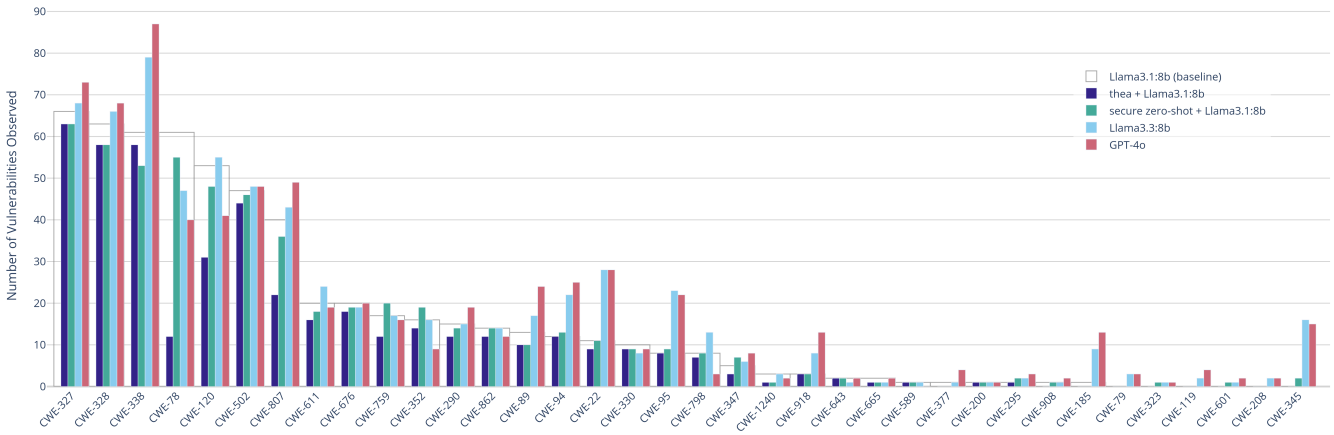

Figure 6: Vulnerabilities observed using each approach (lower is better). The gray outline shows the vulnerabilities output by Llama 3.1:8b. Thea consistently produces less vulnerabilities than the considered approaches.

concept of using weak hashes *only in the context of storing sensitive information*. We find that this result contrasts with what we would expect from a human developer, for whom the concept of using a weak hash function would not depend on how it is used.

Ultimately, the results we presented here suggest that LLMs learn how to produce different CWEs differently. For some vulnerabilities (e.g. CWE-120), the model learns a concept that represents a deep understanding of how they can occur. For others (e.g. CWE-327), we find that the model learns very specific concepts, representing only one of many different ways in which they can occur.

In Figure 6, we compare the performance of Thea and other approaches across different CWEs. Specifically, we measure the number of observed vulnerabilities after using each approach. We observe that Thea consistently produces less vulnerabilities than the other approaches, and that Thea never introduces new vulnerabilities, unlike GPT-4o and Llama 3.3.

5.2.2 ***RQ2****: What is the impact of Thea on model performance?* To assess the impact of Thea on model performance, we consider only features for CWE-89, CWE-120, CWE-185, CWE-290, CWE-328, CWE-347, CWE-377, CWE-676, CWE-807, and CWE-1240. This is because these are the vulnerabilities that the model seems to learn in a general sense (i.e. not specific instances as separate concepts).

Using Thea reduces the model's initial ability to solve 77.2% of MMLU to 76.3% (a drop of 0.9%), and the 71.9% on InstructHumanEval to 70.1% (a drop of 1.8%).

Unsurprisingly, this demonstrates that Thea has very limited impact on model utility. This is because the identified features are generally monosemantic, i.e. each feature represents one concept well (due to the design of the SAE and our feature identification step), as well as the design of our *activation editing* step. We measure the impact of the activation editing step in RQ4 to validate this claim.

5.2.3 ***RQ3****: How effective is Thea at detecting vulnerabilities in generated code?* To measure the effectiveness of Thea at detecting vulnerabilities, we use the BigCodeBench dataset [46] previously described. Out of the 1066 prompts in BigCodeBench, we observe **193 instances of CWE-338**, 6 instances of CWE-328, 5 instances of CWE-798, 5 instances of CWE-78, and one instance of CWE-502.

We focus only on CWE-338, since the other CWEs have a very small number of occurrence (and so the evaluation would be noisy). We follow the same approach outlined in RQ1, where we simply check if the observed feature value is closer to the safe or unsafe distribution. We consider the feature identified in RQ1 for CWE-338.

	Predicted Safe	**Predicted Unsafe**
Actual Safe	781	75
Actual Unsafe	41	169

Table 3: Confusion Matrix for CWE-338 Detection

In Table 3, we show the confusion matrix of this detection. Overall, we achieve an **accuracy of 89.12%**, an **f1-score of 74.45%**, a **precision of 69.26%**, and a **recall of 80.48%**.

With an accuracy of 89.12%, the identified feature performs well in distinguishing between safe and unsafe code. The recall of 80.48% indicates that it successfully identifies most unsafe code samples, minimizing the risk of overlooking security vulnerabilities. However, its precision of 69.26% suggests that some safe code samples are incorrectly flagged as unsafe.

In RQ1, we demonstrated that some features represent a general concept (e.g. mismanaging a buffer), while others are more specific (e.g. using a weak hash *specifically* to store a password). We hypothesize that the reason behind the misclassifications here is that the feature we identified for CWE-338 is overly specific and not general.

To validate this hypothesis, we measured the coverage for CWE-338: specifically, given all the samples in CyberSecEval that are vulnerable to this CWE (61), we measured how many features are needed to classify 95% of them as safe or unsafe. The result was 10 total features. Doing the same for the other CWEs confirms our hypothesis: most of the CWEs (e.g. CWE-89) can achieve full coverage with 1 feature each. The explanation for the misclassifications, therefore, is that CWE-338 can be captured by studying the

(non-trivial) interaction between multiple features. We leave this exploration for future work, since in this work we set out to study individual features and how they relate to different vulnerabilities.

However, the results here are still promising. Even though CWE-338 is represented by multiple features, and therefore we cannot use our technique to avoid its occurrence too well (from Table 2, we can only avoid 3/61 instances), we can still predict its occurrence reasonably well.

5.2.4 ***RQ4****: How does the activation editing strategy contribute to* Thea*'s overall effectiveness?* This research question is effectively an ablation study, meant to measure the impact of the activation editing technique we proposed in Section 4.3. Several works also leverage SAEs to change or improve model behavior, and the replacement activations they compute differ from ours. [35] use the average value of the desired (safe) distribution; [14, 38] use -200, -50, and zero. We consider all these approaches and compare them to ours for security (RQ1) and functionality (RQ2).

On the security front, our approach performed similarly to -200, -50, and zero, avoiding 15% of the vulnerabilities. Using the average value of each safe distribution resulted in a smaller improvement, i.e. only 5% of the vulnerabilities.

For functionality, the results were not as close. Our approach preserved the most functionality, resulting in 115/164 tasks of InstructHumanEval, and 76.3% on MMLU. Using -200 and -50 proved to be the worst choice, leading to 63/164 tasks on InstructHumanEval and 61% on MMLU. The average value approach resulted in 110/164 for InstructHumanEVal and 68.7% on MMLU. Finally, using zero led to 109/164, and 71% on MMLU. Overall, the results we presented here confirm the intuition behind the activation editing strategy that we proposed: avoiding out-of-distribution values leads to better performance, especially for coding (InstructHumanEval) tasks.

6 Related Work

We propose a novel repair methodology that leverages sparse autoencoders to identify and control vulnerability-producing features in LLMs, offering a more interpretable and computationally efficient technique to improving the security of automatically generated code. Recent studies in mechanistic interpretability have identified specific model components responsible for key tasks. For example, induction heads facilitate in-context learning by copying relevant information from the input [32], while other works have mapped circuits responsible for indirect object identification [43] and modular arithmetic [27]. While these studies provide fundamental insights into model behavior, they primarily focus on understanding model internals rather than applying this knowledge to practical challenges. Our approach extends this direction by using SAEs to extract and manipulate features that contribute to insecure code generation.

On the other hand, there have been several efforts at editing LLMs to improve their performance, primarily on natural language tasks. While these efforts typically aim to modify factual associations within LLMs [26], the work done in [22] attempts to apply model editing techniques to improve the correctness of the code generated by LLMs. Their approach focuses on modifying the LLM to internalize the correct output for previously incorrect generations. These methods typically involve identifying and modifying individual neurons responsible for specific knowledge, but their reliance on neuron-level interventions makes them susceptible to superposition, where multiple features are entangled. This limitation reduces their applicability to complex tasks such as ensuring code security. In contrast, our approach circumvents these issues by disentangling security-relevant features using SAEs, allowing for precise and scalable modifications to reduce the likelihood of generating vulnerable code. Moreover, by targeting underlying concepts that contribute to vulnerabilities rather than specific code instances, our approach avoids overfitting and promotes generalizable security improvements.

Closer to our approach, some works have investigated *activation steering* to control the behavior of LLMs using SAEs. For example, [29] investigated using SAEs to avoid jailbreaks. [14] investigated using SAEs for unlearning. These approaches typically involve computing *steering vectors* by extracting the average value of all features from samples of desired behaviors, or by suppressing (using negative numbers or zero) the activation of problematic features. Instead of applying uniform transformations like these works, Thea first identifies relevant features for modification and adaptively adjusts their activations, enabling more targeted and effective control. We directly compared our approach to activation editing to theirs, and our results demonstrate the advantage of our adaptive technique.

7 Discussion

We presented a novel technique to identify concepts an LLM leverages at inference and show how these concepts produce different kinds of vulnerabilities. We introduced Thea, a tool that identifies the use of these concepts and repairs model executions by removing the LLM's access to these vulnerability-producing concepts. We demonstrated the efficacy of Thea in improving the security of automatically generated code. In our empirical evaluation, we demonstrated that newer LLMs can produce more vulnerable code than older LLMs.

Our technique leverages static analysis to identify insecure code. While static analysis can lead to false reports, we believe that a more robust mechanism for detecting vulnerabilities would only strengthen our results. We did not compare Thea to program repair techniques, since our goal is to allow the model to produce more secure code to begin with. Importantly, Thea can be combined with program repair to enhance the security of automatically generated code. Finally, our analysis revealed that LLMs learn to produce CWEs in different ways. For some vulnerabilities, such as CWE-120, the model appears to learn a general concept, representing an understanding of the vulnerability that abstracts over individual insecure patterns. For others, such as CWE-327, the model seems to learn concepts that represent very specific insecure patterns that would cause the vulnerability to occur. By providing insight into how LLMs generate vulnerabilities, our approach can inform the development of more secure models or training strategies.

Acknowledgments

This research is partially supported by the National Research Foundation, Prime Minister's Office, Singapore, under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] [n. d.]. Semgrep — Find Bugs and Enforce Code Standards. https://semgrep.dev/. Accessed: 2024-10-04.

[2] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. 2018. Sanity checks for saliency maps. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 9525–9536.

[3] Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, David Molnar, Spencer Whitman, and Joshua Saxe. 2024. CyberSecEval 2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models. *CoRR* abs/2404.13161 (2024).

[4] Tolga Bolukbasi, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam Tauman Kalai. 2016. Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings. In *NeurIPS*.

[5] Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. 2023. Towards Monosemanticity: Decomposing Language Models With Dictionary Learning. *Transformer Circuits Thread* (2023). https://transformer-circuits.pub/2023/monosemantic-features/index.html.

[6] Nick Cammarata, Gabriel Goh, Shan Carter, Ludwig Schubert, Michael Petrov, and Chris Olah. 2020. Curve Detectors. *Distill* (2020). https://distill.pub/2020/circuits/curve-detectors.

[7] Sviatoslav Chalnev, Matthew Siu, and Arthur Conmy. 2024. Improving Steering Vectors by Targeting Sparse Autoencoder Features. *CoRR* abs/2411.02193 (2024). doi:10.48550/ARXIV.2411.02193

[8] Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. Towards Automated Circuit Discovery for Mechanistic Interpretability. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[9] Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2025. A Comprehensive Study of LLM Secure Code Generation. *CoRR* abs/2503.15554 (2025).

[10] Ann-Kathrin Dombrowski, Maximillian Alber, Christopher Anders, Marcel Ackermann, Klaus-Robert Müller, and Pan Kessel. 2019. Explanations can be manipulated and geometry is to blame. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bb836c01cdc9120a9c984c525e4b1a4a-Paper.pdf

[11] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *ICSE*. 81:1–81:13.

[12] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. 2022. Toy Models of Superposition. *Transformer Circuits Thread* (2022). https://transformer-circuits.pub/2022/toy_model/index.html.

[13] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2021. A Mathematical Framework for Transformer Circuits. https://transformer-circuits.pub/2021/framework/index.html.

[14] Eoin Farrell, Yeu-Tong Lau, and Arthur Conmy. 2024. Applying Sparse Autoencoders to Unlearn Knowledge in Language Models. In *Neurips Safe Generative AI Workshop 2024*. https://openreview.net/forum?id=i4z0HrBiIA

[15] Javier Ferrando, Oscar Balcells Obeso, Senthooran Rajamanoharan, and Neel Nanda. 2025. Do I Know This Entity? Knowledge Awareness and Hallucinations in Language Models. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=WCRQFlji2q

[16] Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. 2025. Scaling and evaluating sparse autoencoders. In *ICLR*.

[17] Jingxuan He and Martin Vechev. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *SIGSAC*. 1865–1879.

[18] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[19] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *ICLR*.

[20] Robert Huben, Hoagy Cunningham, Logan Riggs Smith, Aidan Ewart, and Lee Sharkey. 2024. Sparse Autoencoders Find Highly Interpretable Features in Language Models. In *ICLR*.

[21] Kosuke Imai, Luke Keele, and Dustin Tingley. 2010. A general approach to causal mediation analysis. *Psychological methods* 15, 4 (2010), 309.

[22] Xiaopeng Li, Shangwen Wang, Shasha Li, Jun Ma, Jie Yu, Xiaodong Liu, Jing Wang, Bin Ji, and Weimin Zhang. 2025. Model Editing for LLMs4Code: How Far are we?. In *ICSE*. 937–949.

[23] Zachary C. Lipton. 2018. The mythos of model interpretability. *Commun. ACM* 61, 10 (Sept. 2018), 36–43. doi:10.1145/3233231

[24] Monte MacDiarmid, Timothy Maxwell, Nicholas Schiefer, Jesse Mu, Jared Kaplan, David Duvenaud, Sam Bowman, Alex Tamkin, Ethan Perez, Mrinank Sharma, Carson Denison, and Evan Hubinger. 2024. *Simple probes can catch sleeper agents*. https://www.anthropic.com/news/probes-catch-sleeper-agents

[25] et al. McGrath. 2024. Understanding and Steering Llama 3 with Sparse Autoencoders. *Goodfire Research* (2024). https://www.goodfire.ai/papers/understanding-and-steering-llama-3

[26] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and Editing Factual Associations in GPT. In *NeurIPS*.

[27] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. 2023. Progress measures for grokking via mechanistic interpretability. In *ICLR*.

[28] Yao Ni, Shan Zhang, and Piotr Koniusz. 2025. PACE: marrying generalization in parameter-efficient fine-tuning with consistency regularization. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1958, 29 pages.

[29] Kyle O'Brien, David Majercak, Xavier Fernandes, Richard Edgar, Jingya Chen, Harsha Nori, Dean Carignan, Eric Horvitz, and Forough Poursabzi-Sangdeh. 2024. Steering Language Model Refusal with Sparse Autoencoders. *CoRR* abs/2411.11296 (2024).

[30] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. 2020. Zoom In: An Introduction to Circuits. *Distill* (2020). doi:10.23915/distill.00024.001 https://distill.pub/2020/circuits/zoom-in.

[31] Bruno A. Olshausen and David J. Field. 1997. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research* 37, 23 (1997), 3311–3325.

[32] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2022. In-context Learning and Induction Heads. *Transformer Circuits Thread* (2022). https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

[33] OpenAI. 2022. ChatGPT: OpenAI's Conversational Language Model. https://openai.com/chatgpt.

[34] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768.

[35] Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Matt Turner. 2024. Steering Llama 2 via Contrastive Activation Addition. In *ACL*. 15504–15522.

[36] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).

[37] Benjamin Steenhoek, Kalpathy Sivaraman, Renata Saldivar Gonzalez, Yevhen Mohylevskyy, Roshanak Zilouchian Moghaddam, and Wei Le. 2025. Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection Repair in the IDE . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–13. doi:10.1109/ICSE55347.2025.00126

[38] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. 2024. Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet. *Transformer Circuits Thread* (2024). https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html

[39] Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhiff, and Riccardo Scandariato. 2024. Prompting Techniques for Secure Code Generation: A Systematic Investigation. *CoRR* abs/2407.07064 (2024).

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal

Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[42] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval. *CoRR* abs/2407.02395 (2024).

[43] Kevin Ro Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. 2023. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 Small. In *ICLR*.

[44] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *ICML*.

[45] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. In *ICLR*.

[46] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, and et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *ICLR*.

Received 14 March 2025; revised 18 July 2025; accepted 17 October 2025