PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs)

Mahmoud Nazzal
mn69@njit.edu
New Jersey Institute of Technology
Newark, NJ, USA

Issa Khalil
ikhalil@hbku.edu.qa
Qatar Computing Research Institute
Doha, Qatar

Abdallah Khreishah
abdallah@njit.edu
New Jersey Institute of Technology
Newark, NJ, USA

NhatHai Phan
phan@njit.edu
New Jersey Institute of Technology
Newark, NJ, USA

Abstract

The capability of generating high-quality source code using large language models (LLMs) reduces software development time and costs. However, recent literature and our empirical investigation in this work show that while LLMs can generate functioning code, they inherently tend to introduce security vulnerabilities, limiting their potential. This problem is mainly due to their training on massive open-source corpora exhibiting insecure and inefficient programming practices. Therefore, automatic optimization of LLM prompts for generating secure and functioning code is a demanding need. This paper introduces PromSec, an algorithm for prompt optimization for secure and functioning code generation using LLMs. In PromSec, we combine 1) code vulnerability clearing using a generative adversarial graph neural network, dubbed as gGAN, to fix and reduce security vulnerabilities in generated codes and 2) code generation using an LLM into an interactive loop, such that the outcome of the gGAN drives the LLM with enhanced prompts to generate secure codes while preserving their functionality. Introducing a new contrastive learning approach in gGAN, we formulate the code-clearing and generation loop as a dual-objective optimization problem, enabling PromSec to notably reduce the number of LLM inferences. As a result, PromSec becomes a cost-effective and practical solution for generating secure and functioning codes.

Extensive experiments conducted on Python and Java code datasets confirm that PromSec effectively enhances code security while upholding its intended functionality. Our experiments show that despite the comprehensive application of a state-of-the-art approach, it falls short in addressing all vulnerabilities within the code, whereas PromSec effectively resolves each of them. Moreover, PromSec achieves more than an order-of-magnitude reduction in operational time, number of LLM queries, and security analysis costs. Furthermore, prompts optimized with PromSec for a certain LLM are transferable to other LLMs across programming languages and generalizable to unseen vulnerabilities in training. This study presents an essential step towards improving the trustworthiness of LLMs for secure and functioning code generation, significantly enhancing their large-scale integration in real-world software code development practices.

CCS Concepts

• **Security and privacy → File system security**; **Software security engineering**; **Vulnerability scanners**; *Formal security models*; *Domain-specific security and privacy architectures*;

Keywords

LLMs, code generation, secure and functioning codes, graph generative adversarial networks.

ACM Reference Format:
Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs). In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690298

1 Introduction

Large language models (LLMs) are at the forefront of machine learning (ML), demonstrating exceptional performance in various natural language processing (NLP) tasks. These models, including the generative pre-trained transformer (GPT) series by OpenAI [49, 6], utilize a large-scale, transformer-based architecture [61] trained on diverse NLP datasets. Recently, LLMs have attracted attention for generating high-quality source code, significantly benefiting software development by reducing time and expertise costs [75]. Their growing popularity is evident, with tools like GitHub's Copilot attracting over a million paid subscribers across 37,000 organizations [19]. The effectiveness of these LLMs stems from their extensive training using vast source code databases. However, this reliance on open-source training data introduces significant security risks [54]. The LLMs often replicate security flaws present in their training examples, leading to vulnerabilities in the generated code [43, 25, 8, 53, 45, 51]. This issue raises concerns about the reliability of LLMs for secure code generation, also known as program synthesis, and highlights the necessity to address and mitigate these inherent security vulnerabilities effectively.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690298

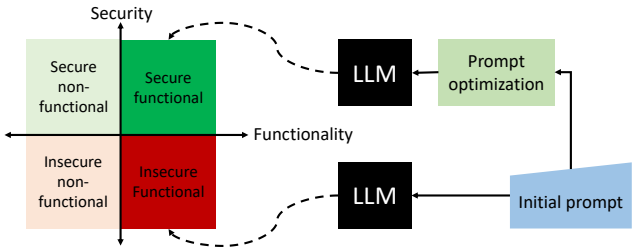


Figure 1: With prompts from average users, LLMs tend to generate codes with security vulnerabilities.

Challenges. The generation of secure and functionally correct code with LLMs faces the fundamental challenge of prompt optimization. Significant research efforts [48, 74, 46, 69, 36] have been directed towards engineering and optimizing LLM prompts for specific applications, highlighting the vast prompt search space and developing heuristics to navigate this space effectively. Similarly, in code generation, a recent state-of-the-art (SoTA) approach [44] shows the diversity in security and functionality of code generated with respect to variations in hand-crafted prompts. However, the search space challenge is more severe in the code generation application due to the need for external tools to assess code security and functional fidelity, making prompt optimization more complex. Moreover, the lack of differentiability of LLM prompts with respect to these security and functionality metrics, coupled with the absence of intuitive heuristics for maintaining security and functionality, adds to the challenge. Also, as detailed in Section 4, the one-to-many nature of function-code implementation further complicates the application of traditional, data-driven security solutions, particularly those based on a supervised learning approach. PromSec addresses these challenges using a graph generative adversarial network (gGAN) model, trainable on a differentiable contrastive loss that synergizes security reduction while upholding intended functionality. As elaborated in Section 4, this model effectively guides the LLM in generating better prompts that enhance security without compromising functionality. The model's effectiveness is demonstrated through an iterative interaction loop with the LLM, progressively fine-tuning the generation process to meet these dual objectives.

Prior Approaches and Limitations. As pictorially depicted in Fig. 1, LLMs are inclined to produce functional and insecure code bases if one depends on regular prompts by average users. This is especially the case whenever the intended code's application has any security notation, as we empirically validate in Section 3. Efforts in prompt engineering for secure code generation, such as hand-crafted templates by Pearce et al. [44], are exhaustive and time-consuming. Methods like bounded model checking (BMC) [8] are restricted to memory access issues. Other works [29, 28] apply reinforcement learning to improve the security of LLM-generated code. However, these operate directly on source code rather than prompts. The only effort attempting prompt optimization is in [23], but it requires white-box access, limiting its applicability to proprietary models like GPT and Bard. Overall, these approaches fall short of offering a unified framework for optimizing prompts for security and functionality, as depicted in Fig. 1.

PromSec Overview. Based on the previous discussion, we formulate prompt optimization in code generation as a two-objective optimization problem aiming at code security and functionality preservation. To meet these objectives despite the non-differentiable nature of prompts with respect to their quantifications, we follow an iterative approach where we improve the code and then improve the prompt in each iteration. Code improvement is achieved by utilizing a gGAN model specially designed for this purpose; given an input code, it generates an output code with semantic similarity and reduced vulnerability. The model is trained to do so by minimizing a novel (differentiable) contrastive loss that involves the number of common weakness enumerations (CWE)[1] as defined by MITRE [57] and the graph embedding distance. The LLM is then used to produce an improved prompt based on the improved code. In essence, the key difference this approach has over existing methods is the use of actual code fixes to steer the generation of improved prompts, rather than leaning on diagnostic tools such as what is done in [44, 8]. Our approach is anticipated to guide prompts toward a more thorough exploration of the solution space, as evidenced in performance assessments where vulnerabilities are swiftly resolved with minimal iterations at a low cost. Our focus is on fixing identified vulnerabilities, rather than vulnerability detection which is a distinct problem.

Summary of Contributions. The contributions in this paper are as follows.

- Introduction of PromSec, an algorithm for automatically optimizing prompts in LLM to generate secure source code while upholding its intended functionality.
- Development of a graph generative neural adversarial network (gGAN) model, characterizing semantic-preserving, and security-enforcing source code fixing as a two-objective optimization problem. This model, trained by minimizing a novel contrastive loss function, enables the application of semantic-preserving security fixes to the code graph.
- Demonstrating the efficacy of PromSec through comprehensive empirical validation, which confirms its capability in enhancing security and functionality in code generation using LLMs, and transferability across different LLMs, CWEs, and programming languages.

Paper Outline. Section 2 presents relevant preliminaries. An empirical study of the need for code securing with LLMs is articulated in Section 3. Section 4 details the proposed PromSec algorithm. Experiments and results are presented in Section 5. Section 6 summarizes related work. A discussion is provided in Section 7 with the conclusions in Section 8.

2 Background

Code Generation with LLMs. The idea of code generation traces back to sequence-to-sequence models, such as recurrent neural networks (RNN) and long short-term memory (LSTM). Here, the model takes a user's natural language prompt that represents their intent and produces a sequence of tokens representing the desired

[1] Common weakness enumerations (CWEs) are standardized categories that identify common software weaknesses and vulnerabilities. These are annually released by MITRE [57], serving as a comprehensive reference to help organizations understand, identify, and address security issues in software.

Table 1: Summary of major topics covered by our 50 prompts.

Topic	Description
Database Interaction	Interacting with databases, searching, deleting records, and data retrieval.
Security and Encryption	Data and file security, encryption, password hashing, and API key management.
Network and System Management	Network diagnostics, remote system monitoring, and IoT device control.
User Authentication	User login features and authentication with Python Flask.
File and Data Processing	Custom file managers, file operations, and user data processing.
Web Scraping and Web Apps	Web scraping, rendering HTML content, and automating web-related tasks.
Mathematical and Data Analysis	Calculators, data processing tools, and mathematical operations.

code. Subsequently, Transformer models [61], and later LLMs [49, 6, 41, 12, 21] have contributed to better code generation. Today, LLMs have gained prominence in this field. Trained on extensive datasets encompassing various programming languages and patterns, LLMs excel in producing syntactically correct, logically consistent, and functioning source code in many programming languages [9, 3]. Their use extends beyond code generation to debugging, code translation, and even full software application development [40, 9]. As a result, employing LLMs can boost productivity by 30% to 50% in software development and reduce the costs of Information Technology (IT) by up to 65% [58, 26, 14]. However, concerns remain about their ability to generate secure and vulnerability-free code consistently [43, 25, 8, 53, 45, 51]. The existence of such vulnerabilities in code can enable malicious exploitation, potentially leading to cyberattacks, data breaches, and system instability. Code vulnerabilities, including buffer overflow, injection attacks, and authentication bypass, can lead to unauthorized access and data theft [7, 5].

Graph Representation of Source Code. Graph representation of code abstracts relationships among different program elements and serves as input for a wide range of software analysis tasks [64, 37, 1, 16, 65]. The three widely used types of graph representations of code are abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs) [30]. Appendix A.1 illustrates constructing these graphs from a given code example. ASTs provide a hierarchical and structured representation of the syntactic elements within a program, enabling in-depth examination of code syntax and organization [67]. CFGs, on the other hand, capture the control flow within a program, representing how control is transferred between different code blocks and aiding in understanding program execution paths [72]. DFGs represent data dependencies and the flow of information within the program, offering insights into data-centric aspects of code behavior.

Graph-based code repair has emerged as a useful tool mainly applied for detecting malfunctioning portions of source code (patches) and suggesting remedies to them. As leading graph representation models, GNNs can effectively capture the complex relationships and dependencies inherent in programming languages [1, 16]. In the context of automatic code repair, GNNs are mainly used to identify buggy portions of source code thereby hinting at potential fixes [60, 11]. These fixes can range from minor syntax corrections to significant structural changes aimed at closing security gaps.

GNNs and Generative GNNs. Graph neural networks (GNNs) extend deep learning to graph data, utilizing neural network layers for message passing and aggregation [66]. Efficiently transforming graph information into node embeddings, GNNs excel in capturing complex relationships within graphs, leading to SoTA performance in various applications [68]. Among generative GNNs, graph generative adversarial networks (gGANs) and graph variational autoencoders (gVAEs) are particularly noteworthy. A gGAN [63] employing adversarial frameworks consists of a *generator* and a *discriminator*, for complex network synthesis. The generator creates graphs, while the discriminator assesses their authenticity. Conversely, a gVAE [31] uses a variational autoencoder architecture, encoding graph data into a latent space and then decoding it to generate new graphs.

Contrastive Learning. Contrastive learning is a model training technique that emerged as a remedy for the data-labeling requirement of supervised learning. Instead of using specific data labels, a contrastive learning scheme identifies positive and negative data pairs. A positive pair comprises two data points of relatively strong similarity as opposed to a negative pair. Therefore, the model is trained to minimize the differences between items in a positive pair and maximize it for items in a negative pair. A clear advantage of this learning scheme is alleviating the need for data labels explicit supervision [10, 24]. Contrastive learning has shown notable advantages in tasks like image and text representation [10], and its applicability extends to the domain of GNNs. In GNNs, specifically generative GNNs, contrastive learning facilitates the learning of node and graph embeddings that accurately capture the underlying structure and features of graph data [62, 70, 65]. As a result, contrastive learning and generative GNNs are particularly suitable for tasks where understanding the relationships and structures within data is crucial, such as in social network analysis, recommendation systems, and bioinformatics.

3 How Secure are LLMs in Code Generation?

Since LLMs are commonly perceived to generate code bases prone to security vulnerabilities [43, 25, 8, 53, 45, 51], our initial focus is to assess this concern on modern LLMs. To achieve this objective, we conducted the following experiment. We manually design 50 prompts, each asking the LLM to generate a code base that performs a specific task. While tasks have some security notions, these prompts do not explicitly ask the LLMs to beware of security issues, nor do they represent bad programming practices. They are just subtle verbal descriptions of what an LLM is requested to generate, and they are aimed to represent the skill level of an ordinary LLM user without security expertise. Table 1 summarizes the main topics covered by these prompts. These topics cover a wide range of Python scripting and application development areas, including database access, security, encryption, network and system management, user authentication, file and data processing, web scraping, and mathematical and data analysis.

We feed the prompts to a set of prominent LLMs including Google's Bard [21], Meta's CodeLlama-70B-Instruct [13], OpenAI's GPT-3.5 Turbo [41], and OpenAI's GPT4 [42], and observe their generated codes. Then, we analyze the security of the generated

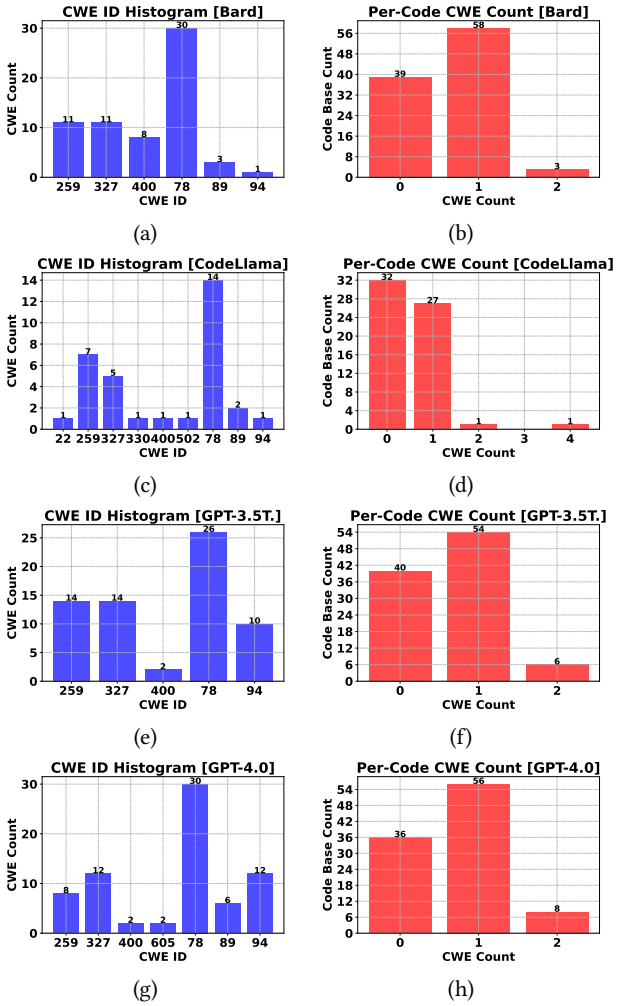


Figure 2: Histogram of CWEs in 100 generated code bases according to the prompts summarized in Table 1 (left) and the histogram of the CWE count per code base (right), for Bard [21], CodeLlama-70B-Instruct [13], GPT-3.5 Turbo [41], and GPT4 [42] row-wise, respectively.

code bases using widely applied tools such as Bandit [15] and SpotBugs [56]. Each prompt is fed to each LLM twice, resulting in 100 generated code bases per LLM. Fig. 2 shows the outcomes of the security analysis across these LLMs presented in rows, respectively. For each LLM (row), we first plot a histogram of the identified CWEs in the generated codes (left) and a histogram of the counts of CWEs in each code base (right). The figure supports previous studies [43, 25, 8, 53, 45, 51] that LLMs generate codes prone to security vulnerabilities as evidenced by the identified CWEs.

We note that CWE 78 stands out as the most frequently recurring security vulnerability among the identified ones. There is a noticeable overlap in the identified CWEs across the four LLMs. Moreover, with Bard, GPT-3.5 Turbo, and GPT-4.0, the security analysis reveals that no more than 40% of the generated code bases are entirely secure, according to the security analysis tools we use for Python and Java codes. It is noted that CodeLlama-70B-Instruct generates code bases in 61 of the times it is prompted. This is consistent with the common belief that Llama models are overly protective and tend to have a high *false rejection rate* [4]. However, the code bases generated still have similar vulnerability occurrences. We emphasize that these findings are derived without prompting the LLMs to explicitly or implicitly create CWEs. CWEs surfaced solely because the LLMs were instructed to generate code for security-related tasks. This experiment illustrates that SoTA LLMs often produce insecure code when tackling security-related tasks. Hence, it is crucial not to unquestionably rely on their outputs.

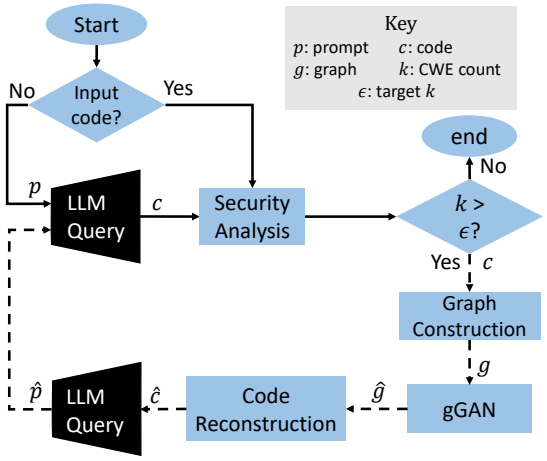


Figure 3: An overview of PromSec's pipeline.

4 PromSec

The key idea behind PromSec is applying contrastive learning within a graph generative adversarial network (gGAN) combined with static security analysis. This unified method aims to direct the LLM in producing secure and functioning code as an alternative to conventional optimization approaches that are hindered by the non-differentiable nature of generating secure and functioning code.

Fig. 3 shows the main steps of our proposed PromSec approach. First, an initial prompt is obtained. Such a prompt is taken directly from the user as a verbal description characterizing her/his intent [32]. The initial prompt is forwarded to the LLM and a code is generated. The generated code is then forwarded to the *Security Analysis* module, which is a static security tool to identify the number of CWEs in the code. Alternatively, if the input is source code c, it is passed directly to the *Security Analysis* module. The code is considered the final output if the security analysis identifies no CWEs, denoted as "$k = 0$" (where 'k' represents the count of identified CWEs by the security analysis tool). Otherwise, the code is transformed into a graph representation g (e.g., AST) using the *Graph Construction* module, then forwarded to a *gGAN*. This gGAN model aims to enhance the input graph representation, striving for a more secure (with reduced CWEs) one, $\hat{g}$, while maintaining the intended functionality. Once generated, the graph $\hat{g}$ is converted back into code $\hat{c}$ through a *Code Reconstruction* module. This reconstructed

code is then reverse-engineered by the LLM[2] to generate a prompt corresponding to $\hat{c}$, $\hat{p}$. The new prompt $\hat{p}$ is fed back into the LLM, generating new code versions in a loop that continues until the CWE count reduces to a small desired value (ϵ), which ideally is zero) or for a prescribed number of cycles.

In this section, we first establish the suitability of a contrastive learning framework for the training of the aforementioned gGAN model. Then, we provide the architectural details and the proposed contrastive loss to be used in the training of this model along with its training algorithm.

4.1 Code Fixes are One-to-Many

Function-implementation relationship in code bases is well-known to be a one-to-many mapping, i.e., the same semantic function can be implemented as a code in many ways [33, 59]. Furthermore, code similarity detection systems like Facebook's Aroma and Intel's Machine Inferred Code Semantics Analysis (MISIM) illustrate the ability to recognize different code implementations fulfilling identical functions, supporting the one-to-many mapping [33].

It is interesting to discuss the implication of the one-to-many function-to-code relationship on suitable learning of generative models on code graphs. Let us recall that a code can be represented as a graph. Thus, it is possible to obtain an improved code by utilizing a generative GNN model trained to generate better code graphs. In supervised learning of generative models, model training requires labeled data, where a data point label represents the desired outcome that the model needs to generate. In an application like source code fixing, an immediate approach is to simply feed a generative model with an input code graph and train it to generate a secure version of this code graph. However, recalling the aforementioned one-to-many function-code mapping, it is evident that the same code can have many secure versions. In other words, for a given insecure code snippet, there could be multiple and structurally diverse but functionally equivalent secure versions. Accordingly, there is no notion of a single desired outcome or, equivalently, no notion of a unique data label.

To demonstrate this limitation, we conduct the following experiment. We consider a set of 100 Python codebases, each containing multiple CWEs. For each codebase, we generate three CWE-free *versions*. Specifically, each test codebase is provided to the LLM, which is instructed to generate a secure version that retains the same functionality. This secure version is analyzed using a security tool to ensure it is free of CWEs and is manually verified to maintain the same functionality. This process repeats until three secure versions are obtained per sample. Next, we define and calculate the average *inter-version* distance, which is the mean graph edit distance between the code graph of the original test codebase and that of its corresponding CWE-free versions. We also calculate the average *intra-version* distance, which denotes the average graph edit distance among the graph representations of the CWE-free versions of each codebase. We calculate these metrics for the 100 considered codebases.

Fig. 4 illustrates the results of those two metrics per code base (normalized by the maximum value). It is worth noting that the

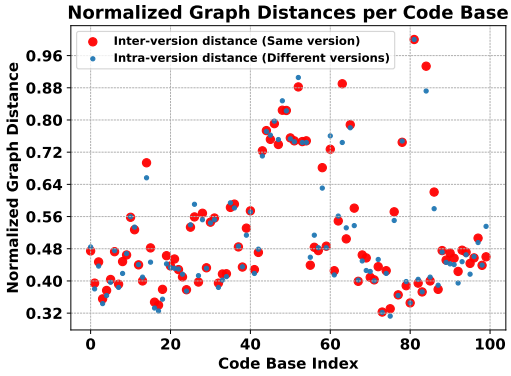


Figure 4: A comparison of the average inter-version and average intra-version graph edit distances per code base.

average inter-version and average intra-version distances are comparable. Furthermore, on many occasions, the inter-version distance is even less than the intra-version distance. This observation indicates that the edit distance between the original code base and its CWE-free versions is less than or at least comparable to the distance among its CWE-free versions.

The observed diversity in secure and functioning code versions suggests the viability of integrating contrastive learning in the gGAN model for code improvement alongside static security analysis. This combination provides an effective alternative approach for guiding LLMs in generating secure and functioning code using optimized prompts. Contrastive learning's strength lies in distinguishing between various examples, making it suitable for scenarios with multiple valid outcomes. It can train models to recognize a broad range of secure coding patterns beyond replicating a single secure version, expanding their understanding of diverse secure coding practices.

4.2 The Training and Operation Pipeline

We formulate prompt optimization for secure and functioning code generation with LLMs as follows:

$$p^* = \underset{p}{\operatorname{argmin}} \left[\alpha k(c) + \beta d(c, c_0)\right], \tag{1}$$

where p is a prompt, p^* is the optimal prompt, c is the code to be generated, c_0 is the original code either given as a code starting point or the code generated with an initial prompt p_0. The term $k(c)$ denotes the CWE count in the code c, and $d(c, c_0)$ is a metric representing the functional discrepancy between c_0 and c. Also, α and β are weight parameters.

As highlighted in the Introduction Section, the essential challenge in this optimization is that p is not differentiable with respect to $k(c)$ and $d(c, c_0)$. Even modeling these functions in terms of p is challenging since it requires modeling the LLM with external security analysis tools. To address this challenge, we approximate the optimal prompt p^* in Eq. 1 by using a gGAN model to generate improved code graphs that lead to improved prompts for generating secure and functioning code in an iterative manner, as follows.

gGAN rectifies code issues by altering its graph representations, which are then translated into prompt adjustments. Operating on

[2]By explicitly instructing the LLM to analyze the codebase and estimate a detailed prompt that could have generated it.

graph data, this model follows the framework akin to the generator-discriminator setup in classical GAN networks. Here, inputs drive the generator to produce outputs meeting specific criteria, while the discriminator evaluates the adherence of these outputs to the defined requirements. As it operates on graph data, it is classified as a GNN-based GAN. Architectures such as the graph convolutional network (GCN)-based GAN [34] can be utilized in this context.

The proposed learning mechanism for the gGAN is outlined in Algorithm 1. We train the model using a unique contrastive loss for the generator, ensuring the generation of secure and functioning code graphs. We formulate the contrastive loss of the gGAN as follows:

The **generator loss** is composed of adversarial and contrastive losses, denoted as $\mathcal{L}_{\text{adv}}$ and $\mathcal{L}_{\text{contrastive}}$ respectively, balanced by a hyperparameter λ as

$$\mathcal{L}_G = \mathcal{L}_{\text{contrastive}} + \lambda \mathcal{L}_{\text{adv}}. \tag{2}$$

- **Adversarial Loss [20].** The adversarial loss $\mathcal{L}_{\text{adv}}$ evaluates how effectively the generator G can deceive the discriminator D. It is defined as

$$\mathcal{L}_{\text{adv}} = -\mathbb{E}[\log D(G(g_i))]. \tag{3}$$

- **Contrastive Loss.** The contrastive loss $\mathcal{L}_{\text{contrastive}}$ is incorporated in terms of the number of CWEs (denoted by k) and an embedding similairty measure S:

$$\mathcal{L}_{\text{contrastive}} = -\log(x), \tag{4}$$

$$\text{s.t. } x = \frac{e(\alpha \Delta k_i + \beta S_i)}{e(\alpha \Delta k_i + \beta S_i) + \sum_{j \neq i} e(\alpha \Delta k_{ij} + \beta S_{ij})}, \tag{5}$$

where $\Delta k_i = k(c_i) - k(\hat{c}_i)$ is the difference in CWE counts between the code c_i represented by an input graph g_i and its fixed counterpart $\hat{c}_i$ represented by the generated graph $\hat{g}_i$, S_i is a graph embedding similarity measure between the two graphs g_i and $\hat{g}_i$, Δk_{ij} is the difference in CWE counts of the code bases represented by graphs g_i and g_j, S_{ij} is their graph similarity, and $e(z) \triangleq \exp(-z)$, α and β are used to control balancing code security and functional similarity, and the summation is over graphs in the training dataset.

The design of the contrastive loss function aims to encourage the generation of graphs with fewer CWEs and greater similarity in embeddings between the original and generated graphs, thereby maintaining code functionality [72, 17, 65, 37].

Discriminator Loss. The discriminator loss $\mathcal{L}_D$ follows the standard GAN format, being the binary cross-entropy loss between real and generated graphs [20], as follows:

$$\mathcal{L}_D = -\mathbb{E}[\log D(g_i)] - \mathbb{E}[\log(1 - D(G(g_i)))]. \tag{6}$$

Gradient Updates. For the discriminator and generator modules, gradients of these loss functions are computed and used for updating the respective parameters through stochastic gradient descent (SGD) [50], as follows:

$$\theta_D \leftarrow \theta_D - \eta \nabla_{\theta_D} \mathcal{L}_D, \tag{7}$$

$$\theta_G \leftarrow \theta_G - \eta \nabla_{\theta_G} \mathcal{L}_G, \tag{8}$$

where η represents the learning rate.

Code-to-Graph Conversion. In the proposed pipeline, code can be easily converted into a graph representation (e.g., AST, CFG, DFG) using off-the-shelf parser packages such as [47, 2]. Similarly, converting an AST graph into code is a straightforward task, achieved by unparsing the AST. However, converting a CFG to code is more complicated.

Algorithm 1 Training of the gGAN Model

Input: Training graphs $g_1, g_2, \ldots, g_n$, number of iterations T
Initialize: generator parameters θ_G and discriminator parameters θ_D
for $t = 1$ to T **do**
  for $i = 1$ to n **do**
    Sample a graph g_i
    Generate an improved graph $\hat{g}_i$ using θ_G
    Update θ_D according to Equation (7)
    Update θ_G according to Equation (8)
  end for
end for
Output: Trained generator and discriminator (θ_G and θ_D)

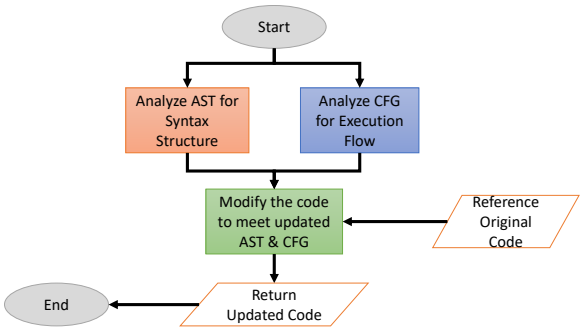


Figure 5: Code reconstruction from CFG graph edits.

To overcome this challenge, we employ a method that considers the initial code, its AST, and CFG graphs alongside the new CFG to reconstruct the updated code as illustrated by the flowchart in Fig. 5. This approach involves modifying elements of the original code based on discrepancies between the CFG and AST graphs of the original and updated versions. These modifications are guided by the need for the new code to align with the new CFG while maintaining consistency. This strategy ensures the integration of the new code changes while upholding the foundational style and structure of the original code. DFG to code conversion follows a process similar to that of CFG.

5 Experiments

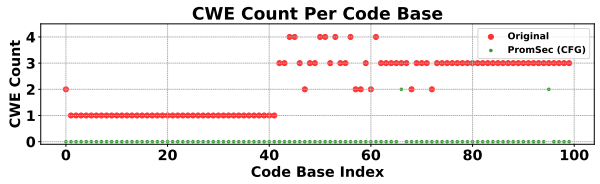
We present here the results of the experiments conducted to evaluate PromSec's performance. In summary, the results validate that PromSec outperforms existing baselines in terms of security, functionality preservation, and various aspects of cost-effectiveness. Importantly, prompt optimizations obtained with PromSec are generalizable to unforeseen CWEs, across different types of LLMs, and different programming languages. The source code and data are available at the link: https://github.com/mahmoudkanazzal/PromSec. In the following subsection, we aim to answer the research questions listed in Table 2.

5.1 The Setup, Dataset, and Baselines

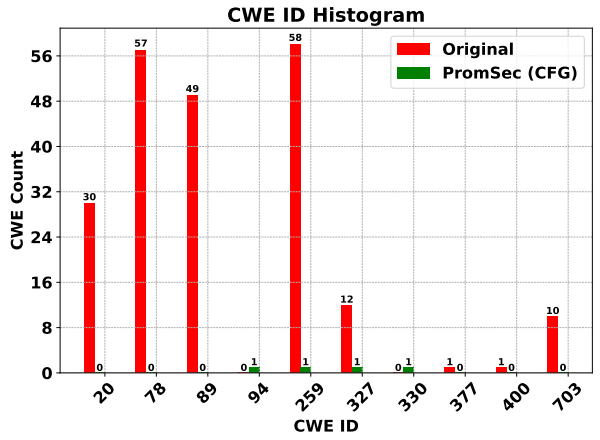
The setup includes an LLM used for code generation. An LLM prompt is a verbal description of the intent for generating the code.

Table 2: A summary of research questions and answers.

Q	Summary	Key Result
1	How successful is PromSec in enforcing code security?	Highly
2	Does PromSec need iterative LLM interaction?	Yes
3	How does PromSec compare to SoTA baselines?	Superior
4	How successful is PromSec in functionality preservation?	Highly
5	What is the impact of code graph types on PromSec?	CFGs perform best
6	Do PromSec prompts transfer to new CWEs?	Yes
7	How is PromSec's transferability across LLMs?	High
8	How is PromSec's transferability across programming languages?	High



(a)



(b)

Figure 6: Performance evaluation plots with CFG-type graphs: (a) per-code base CWE count and (b) the histogram CWEs before and after applying PromSec.

This description is either obtained directly from an assumed user or by instructing the LLM to generate an approximated prompt in case of starting with source code, as detailed in Section 3. Unless stated otherwise, the default LLM used is GPT-3.5 Turbo. The gGAN model used has a Generator with two GraphConv layers and ReLU activations, and a Discriminator with two GraphConv layers with ReLU and Sigmoid activations, implemented using PyTorch Geometric. In our experiments, we set α and β appearing in equations (1) and (5) to 1. We found that increasing α over β slightly enhances security but reduces code functional similarity, and vice-versa. We use two datasets. The first is a Python code base set obtained from [44] and comprising source codes from MITRE's [57] documentation, CodeQL documentation [27], and hand-crafted codes by the authors of [44]. We refer the reader to Appendix A.2 for more details on this dataset. The second is a dataset of Java prompts collected from [22, 71]. For the security analysis, we select Bandit [15] for Python and SpotBugs [56] for Java, for their wide popularity and outstanding performance in terms of detection quality and time

Table 3: Context degree across prompt templates in BLs.

Template	Security Context
1	None (No help)
2	CWE line numbers
3	CWEs IDs
4	CWE line numbers, CWEs IDs
5	CWE line numbers, confidence
6	CWEs IDs, confidence
7	CWE line numbers, CWE IDs, confidence

complexity. However, any other static security tools can be used for this purpose.

We compare the performance of PromSec with the following baselines (BLs):

- **BL1** characterizes Pearce et al.'s work [44] accustomed to code generation. We define 7 prompt templates with increasing context. The contexts of templates are summarized in Table 3. It is because security analysis tools typically return the IDs of identified CWEs, the lines of code where they occur, and the confidence in their detection.
- **BL2** applies BL1 iteratively and at each iteration, greedily selecting the solution with the minimal CWEs count.

We consider the following metrics in our experiments:

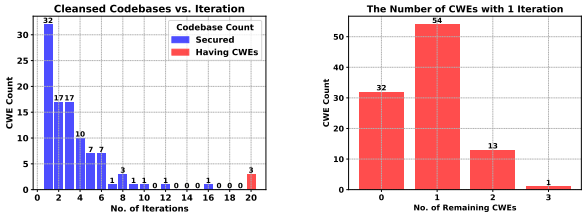
(1) **Security Enhancement.** Quantified by the reduction in the number of CWEs (k) as commonly employed in [43, 44, 53, 55, 23]. This metric is computed based on the outcomes of static security analysis techniques that detect and enumerate CWEs.
(2) **Preservation of Code Functionality.** Quantified by the similarity between code graphs as commonly employed in many works such as [72, 17, 65, 37]. To supplement this, we conduct fuzzing tests [35] on a randomly selected subset of generated code bases, providing an additional layer of evaluation.
(3) **Operational Cost.** Evaluated based on three parameters, including the time taken for code execution (execution time delay), the financial and delay implications of utilizing LLMs, and the costs of static security analysis.

5.2 Performance Evaluation of PromSec

In this subsection, we answer Q1 through Q4 (Table 2) considering CFG as the graph representation. We examine the other two code graph types (AST, DFG) in Subsection 5.3.

First, we address **Q1: How successful is PromSec in code security enforcement?** We train the gGAN model with 500 training graphs. Then, we feed 100 test code bases to PromSec and set the maximum number of iterations to 20. First, with a CFG graph type, Fig. 6(a) shows the number of CWEs for each code base before (denoted by "Original") and after processing with PromSec. Next, Fig. 6(b) presents a histogram of the CWEs in the code bases before and after applying PromSec. It is clearly noticed from both figures that PromSec resolves the majority of CWEs within 20 iterations. In Appendix A.3, we trace the evolution in prompt and code and how a CWE is resolved through a toy example.

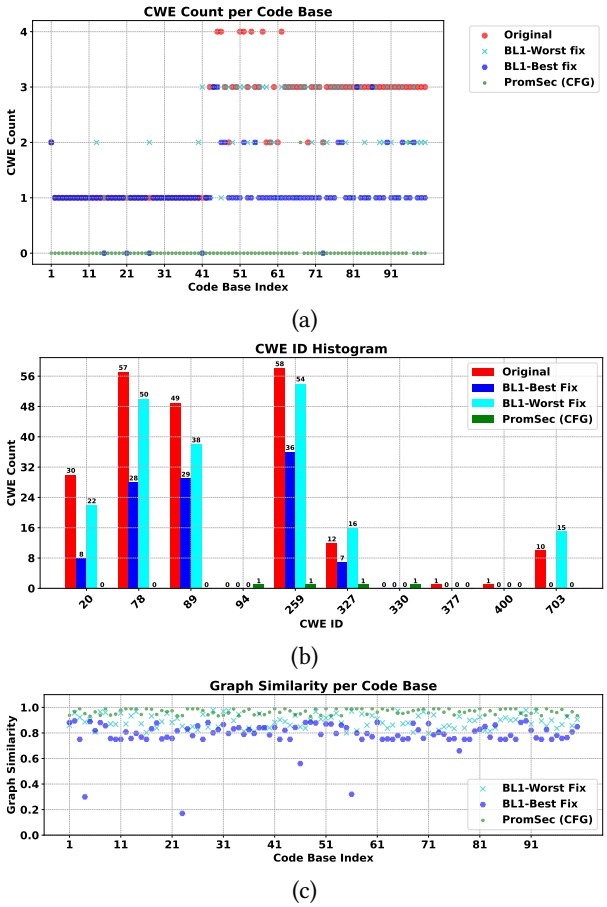
Next, we address **Q2: Does PromSec need iterative LLM interaction?** For this purpose, we plot a histogram of the number of *secured* code bases versus iteration reporting at the last iteration the number of code bases that still have at least one CWE in Fig. 7



(a) (b)

Figure 7: (a) The number of cleansed (fully secured) codebases versus iteration (b) the count of remaining CWEs after the first iteration with CFG graphs.

(a). We also plot a histogram of the count of code bases that still have remaining CWEs after one PromSec iteration in Fig. 7(b). The two histograms confirm that one iteration is not sufficient to secure the majority of code bases (68%). On the other hand, around 49% of the code bases are fully cleaned from CWEs within the first two iterations.



(a)

(b)

(c)

Figure 8: (a) Comparing PromSec and BL1 in terms of per-code CWE counts, (b) remaining CWE distribution, and (c) code graph similarity.

To answer **Q3: How does PromSec perform relative to the baselines regarding security, functionality preservation, and cost?** we replicate the experiment of Q1 and compare PromSec's performance with that of the Baseline 1 (BL1). The results are shown in Fig. 8. The number of CWEs per code base is shown in 8(a), the CWE distribution in 8(b), and the comparison of the graph similarity metric (a proxy for functionality preservation) in Fig. 8(c).

We observe how well BL1 does in the best and worst scenarios in terms of reducing the number of CWEs. In the *best* scenario, we chose the prompt template in BL1 that leads to the fewest CWEs. Conversely, in the *worst* scenario, we chose the one that leads to the maximum CWE count. The results show that PromSec is significantly better than BL1 in resolving CWEs. BL1 does fix some CWEs, but the code it produces is still not completely secure. In terms of preserving code functionality, especially in its best situation, BL1 does not do as well as PromSec. This result shows that PromSec has superior performance both in fixing security bugs (CWEs) as well as upholding the intended code functionality.

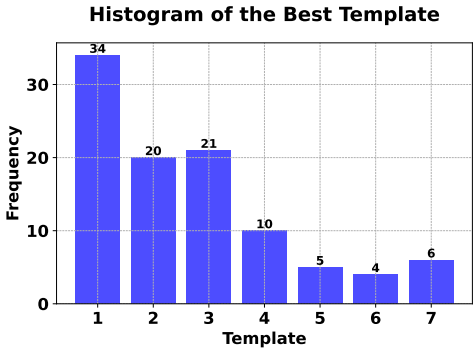
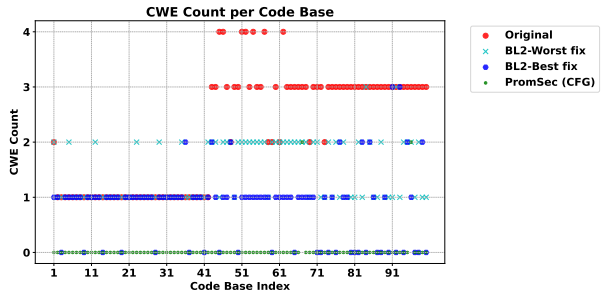


Figure 9: The best template distribution across code bases for BL1. For 34% of the codes, no context is required.

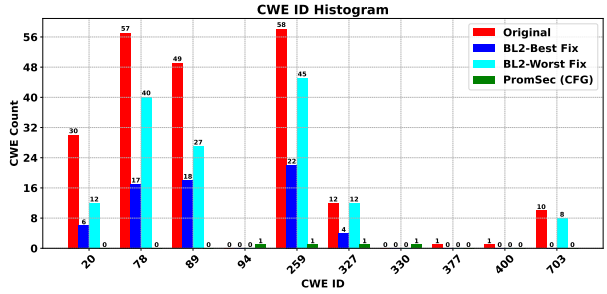
Pearce et al. [44] (BL1) report that adding more context does not always make code completions more secure. To examine this observation in code generation, we show in Fig. 9 how often each prompt template is the most effective for a given test code. Note that we index the templates according to the increase in the context. The histogram in this figure reveals an interesting observation: Template 1, which does not use any security report context, turns out to be the best choice 33% of the time. This result suggests that while including some context is beneficial, adding too much does not always help. In fact, we notice a decrease in preference for templates with more context. This observation is consistent with Pearce et al.'s observation in code completion.

Next, we compare the effectiveness of PromSec against the other baseline, BL2, and present the results in Fig. 10. This figure reveals that BL2 outperforms BL1 in terms of security enhancement, as evidenced by a notably lower count of remaining CWEs. However, this improvement in security enforcement comes at the expense of a reduction in code functionality preservation as shown in Fig. 10(c), where BL2 results in a greater degeneration in code functionality.

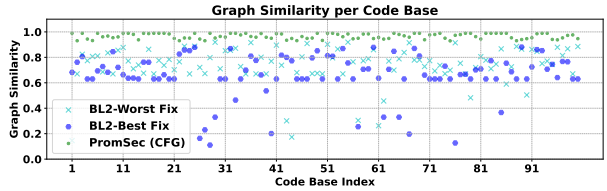
Fuzzing Tests. In this experiment, we address **Q4: How effective is PromSec in preserving the intended code functionality?** In software engineering, fuzzing testing (FT) [35] is a procedure



(a)



(b)



(c)

Figure 10: (a) Comparing PromSec with BL2 in terms of per-code CWE count, (b) remaining CWE distribution, and (c) code graph similarity.

typically used to compare the functionality of two source codes. It involves giving the same inputs to both codes and then measuring differences in their outputs. However, FT usually requires a lot of manual work and can be time-consuming, especially for large numbers of code bases. Also, FT only works well when the code bases have clear and measurable inputs and outputs. Because of these limitations, we use FT as a secondary method to check how well PromSec preserves the original semantics and function of code while improving its security. We compare PromSec with the BL1 and show the results in Table 4.

For conducting FT, we use 20 specially designed code bases that can be tested with fuzzing. Each code base has a main function with specific input and output parameters and calls several internal functions. This setup allows us to quantify any changes in the outputs caused by PromSec. We repeat these tests for 1,000 trials. In each trial, we sample random inputs and calculate the differences in outputs. Then, we calculate the average value of the output differences. An FT is passed if the output changes on average, are smaller than a small threshold value (we set it to 0.01 to account

Table 4: FT comparison of PromSec, BL1, and BL2. Code bases are fully secured by each method, and the number of FT passes is shown.

Initial k	No. of Code bases	No. of Passes		
		PromSec	BL1	BL2
78	10	10	6	3
89	10	10	5	2
Overall	20	20	11	5

Table 5: Empirical cost of BL1, PromSec(1), BL2, and PromSec(20).

Cost Aspect		BL1	PromSec(1)	BL2	PromSec(20)
Time Cost (seconds)	Overall Time	35.62	9.95	649.16	70.92
LLM Query Cost	Number of LLM Queries	7.00	2.00	127.48	14.36
	Input Token Count	1153.10	359.20	22755.80	2359.94
	Output Token Count	574.90	256.00	10348.20	1904.46
	LLM Model Query Time	33.94	9.50	604.92	66.70
Security Analysis Cost	Number of Security Analyses	7.00	1.00	127.48	6.92
	Security Analysis Time	1.652	0.237	29.738	1.756

for minor numeric variations from floating-point calculations, ensuring functional equivalence). Despite this threshold, differences are technically zero in the experiments.

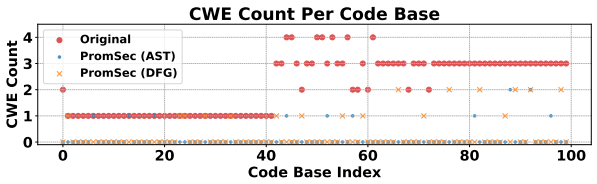
In Table 4 we process a total of 20 code bases that exhibit CWEs 78 and/or 89 with PromSec, BL1, and BL2. Security-wise all these methods fully secure the code bases in the experiment. The table shows that PromSec fully passes the test for all the samples considered. On the other hand, BL1 exhibits some degree of functionality preservation but lower than that achieved by PromSec, whereas BL2 severely fails. This is consistent with the finding on BL2's functionality preservation in the previous experiment.

Empirical Cost Analysis The third aspect of comparing PromSec to the baselines is the operational cost. We compare the costs of BL1 along with one iteration of PromSec denoted by (PromSec(1)), and BL2 with a 20-iteration PromSec (denoted by PromSec(10)). The results are shown in Table 5. As seen in the table, PromSec(1) demonstrates improvements over BL1. The overall time cost is notably lower with PromSec(1), with a mean of 9.95 seconds, in contrast to 70.92 seconds. The LLM query cost category shows a reduction in the number of queries, input and output token counts, and LLM model query time, with PromSec(1) requiring 2 queries versus 7 in BL1, and token counts of 359.2 and 256 compared to 1,153.1 and 574.9, respectively. The query time is also reduced to 9.5 seconds from 33.94 seconds. In the security analysis cost, PromSec(1) requires fewer security analysis queries (1 vs. 7) and less time (0.237 seconds vs. 1.652 seconds). These results indicate that PromSec(1) is more efficient in terms of both time and resource management compared to the existing BL1 baseline. It is noted that this is still the case even though PromSec needs on average 3 iterations, since even with multiplying the cost of PromSec(1) by 3, it is still superior to that of BL1. A more obvious advantage is observed for PromSec(20) over BL2, where PromSec(20) has around an order of magnitude reduction compared with BL2.

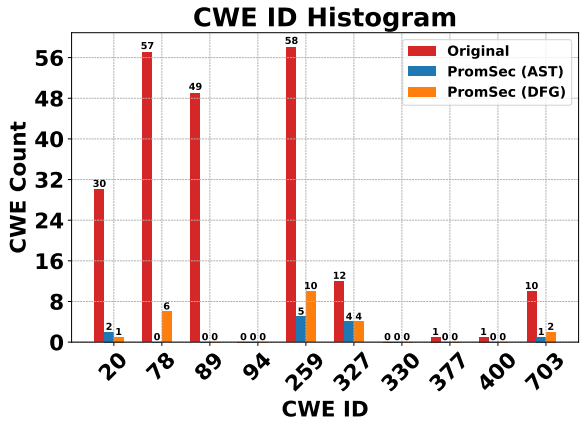
5.3 Different Code Graph Types

Here, we answer **Q5: What is the impact of the code graph type on performance?** We evaluate the performance of PromSec when it utilizes either AST or DFG graph types instead of CFG.

Fig. 11 shows the performance analysis based on AST and DFG graphs. First, consider the AST performance, we note that this performance is acceptable but inferior when compared to the scenario involving CFG graphs. Part of the reason for this is that although ASTs encapsulate the code's syntax, they lack the depth needed to encompass control flow and operational semantics, crucial for a comprehensive security analysis. On the other hand, the performance with DFG-type code graphs is inferior to that of AST-type graphs. Also, both are inferior to CFG graphs. DFGs focus on data flows and dependencies but do not capture the logic and behavior of codes as done by CFGs.



(a)



(b)

Figure 11: Performance evaluation with AST- and DFG-type graphs: (a) per code base CWE count before and after applying PromSec and (b) CWE counts before and after applying PromSec.

5.4 PromSec's Transferability to Unseen CWEs

As shown in our experiments, the ability of the trained gGAN model to fix CWEs is established. Now, we address **Q6: Can PromSec transfer to unforeseen CWEs in its training process?** For this purpose, we conduct the following experiment. We exclude a given test CWE from the training set of the gGAN model and refer to it as a *masked* CWE. Then, we pick test examples that exhibit this masked CWE. In other words, the model does not see any training examples exhibiting the masked CWE.

Fig. 12 depicts the effects of masking specific CWEs, namely, CWE 259, CWE 89, and CWE 78, on PromSec's performance. Several observations can be drawn from this figure. First, masking a given CWE results in a slight decrease in PromSec's effectiveness

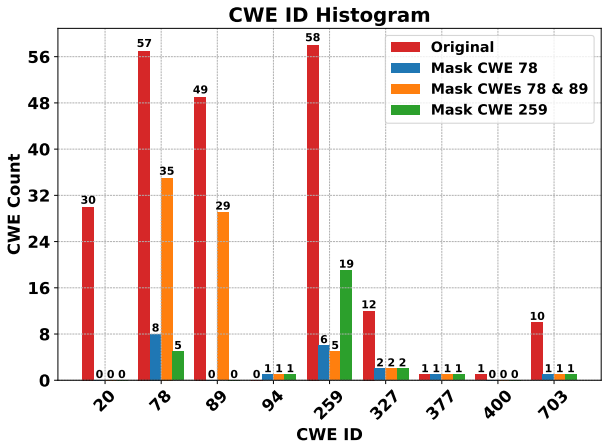


Figure 12: Cross-CWE transferability: The histogram CWEs before and after applying PromSec with masking: CWE 78, both CWE 78 and CWE 89, and CWE 259.

against it and a minor reduction in handling some other related CWEs. This observation validates PromSec's ability to transfer its security-enhancing capabilities to CWEs not directly included in its training. We attribute this transferability to the generalization capability of the gGAN model. Furthermore, it is intuitive to assume that resolving a specific CWE may have a beneficial impact on mitigating related CWEs, especially if they originate from similar underlying issues. For example, prompt optimizations directed to prevent CWE 78 (input injection) might implicitly also mitigate CWE 89 (SQL injection) risks. This result is because both vulnerabilities typically stem from inadequate control of user input, whether it be in operating system commands (in CWE 78) or database queries (in CWE 89). This analysis hints at the possibility of mitigating new security vulnerabilities not explicitly included in the training process of the gGAN.

5.5 PromSec's Transferability across LLMs

Considering the pipeline in Fig. 3, we address **Q7: Are prompt fixes obtained with a given LLM exclusively valid for that LLM or can they be transferable to other LLMs?** For this purpose, we carry out the following experiment. We assume GPT-3.5 Turbo as the base LLM used for optimizing the prompts while we feed these optimized prompts to another LLM to generate the code. We consider Google's Bard and Meta's CodeLlama-13B-Instruct as these LLMs and show the corresponding results in Fig. 13. In both figures, prompts optimized with GPT-3.5 Turbo work well with both Bard and CodeLlama-13B-Instruct. The performance exhibits negligibly small remaining CWEs. This result validates the ability of PromSec's prompt optimizations to transfer across LLMs.

5.6 PromSec's Transferability across Programming Languages

In this subsection we answer **Q8: How successfully can PromSec's optimized prompts generalize across different programming languages?** For this purpose, we collect a dataset of Java prompts

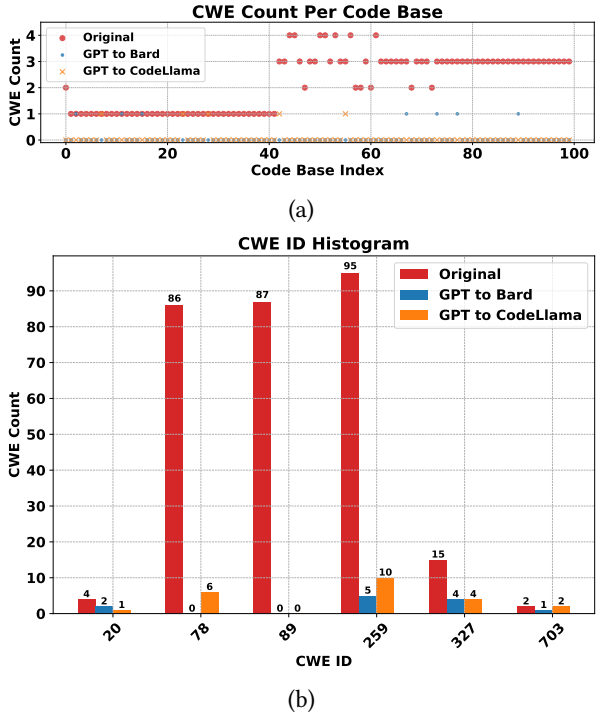


Figure 13: Cross-LLM Transferability: Prompts optimized with GPT-3.5 Turbo are used to generate code with Bard (GPT-to-BARD) and Code Llama (GPT-to-CodeLlama). (a) CWE counts per code base, and (b) CWE counts before and after applying PromSec.

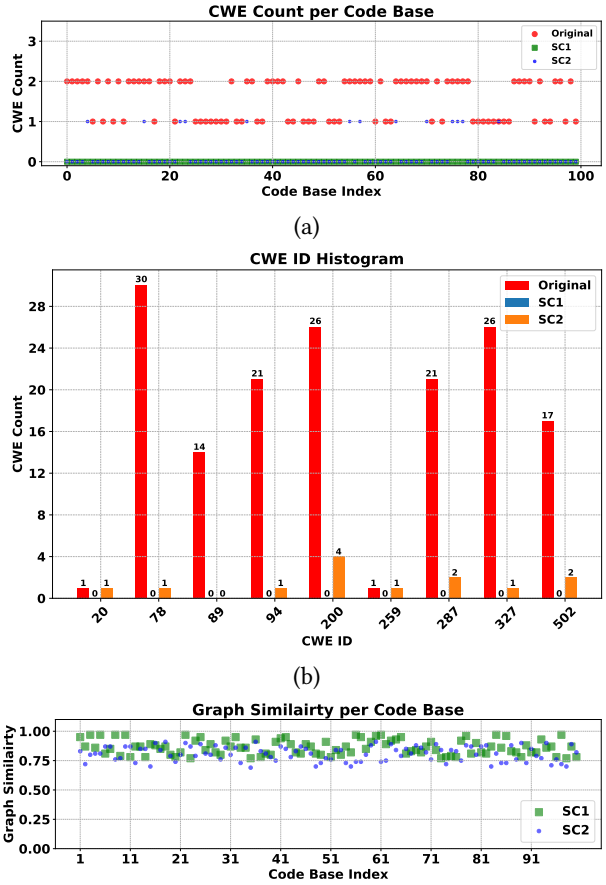


Figure 14: For SC1 (training and testing with Java), and SC2 (training with Python and testing with Java): (a) the CWE count before and after applying PromSec, (b) the CWE ID histograms before and after applying PromSec, and (c) graph similarity.

from [22, 71]. Next, we consider these prompts as inputs to PromSec's pipeline illustrated in Fig. 3. We intentionally employ these prompts to examine the operation of this pipeline starting with prompts, rather than source code as done in the tests considered so far. Next, we train a gGAN model over 200 code bases generated from these prompts. After that, we use the remaining 100 prompts to test the operation of PromSec (we denote this scenario by SC1). To examine the generalization of PromSec across programming languages, we test this prompt dataset with the previous gGAN trained on Python code (we denote this scenario by SC2). Fig. 14 shows a histogram of the CWEs in the test set before and after applying PromSec for these two scenarios in (a) and (b), respectively. Fig14(a) clearly shows that PromSec perfectly resolves the CWEs, similar to the case with Python code experiments. Note that we use the same hyperparameters and number of iterations. Next, Fig14(b) shows almost the same performance is attainable with PromSec despite being trained using codes from another language. The result in Fig14(b) showcases the transferability of PromSec across source code of different programming languages.

5.7 An Ablation Study

To examine the individual roles of the gGAN and the LLM components in the PromSec pipeline, we conduct ablation studies comparing PromSec to two scenarios. In the first scenario (A1), PromSec operates without the gGAN, iterating solely with the LLM. In the second scenario (A2), we exclude the LLM, applying only the gGAN in the pipeline. The outcomes are presented in Fig. 15. As shown, excluding the gGAN significantly impairs the pipeline's ability to reduce CWE counts while maintaining code functionality, indicated by high code graph similarity scores. Conversely, omitting the LLM does not notably affect CWE reduction but significantly undermines code functionality, demonstrated by reduced code graph similarity.

The drop in graph similarity without the LLM is due to the gGAN's lack of contextual understanding. LLMs capture and preserve the code's semantics and functionality, continuously emphasizing its intent. Without LLMs, the gGAN alters code graphs recursively, losing original functionality. LLMs ensure prompts reflect and emphasize the code's functionality and structure, guiding the gGAN to produce precise outputs. These results highlight the interplay between the two components in achieving PromSec's goal: enhancing code security without sacrificing functionality.

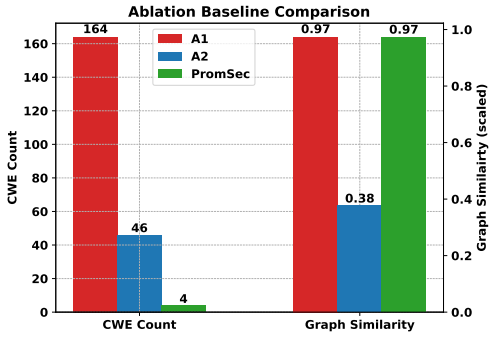


Figure 15: An ablation study comparing CWE counts and code graph similarity across PromSec, scenario A1 (without gGAN), and scenario A2 (without LLM).

6 Related Work

This section briefly revisits prompt optimization for LLMs, especially for secure code generation, and GNNs for code analysis.

Prompt Optimization for General LLM Tasks. Prompt engineering and optimization are crucial in maximizing the potential of LLMs. Works in this area vary by the assumed degree of LLM access and prompt treatment methods. Broadly, they can be divided into two main categories [38]. The first category considers soft prompts. These works consider a continuous prompt space represented as embeddings for the model [48, 39, 52]. This approach involves employing differentiable prompts for these soft embeddings. However, these studies presume familiarity with the latent embeddings of the LLM, leading to limitations in scalability. The second category involves hard prompts and adopts a discrete prompt treatment. This approach starts with an initial prompt and focuses on prompt engineering or optimization by generating a candidate set of feasible prompts and selecting the most suitable one. Examples along this line include the use of Monte-Carlo sampling by Zho et al. [74], gradient-based and beam search by Pryzant et al. [46], and genetic algorithm by Xu et al. [69]. However, these methods, while accomplishing some success, remain computationally intensive and lack assured convergence guarantees. Li et al. [36] offer another instance where reinforcement learning agents dynamically enhance provided prompts within textual LLM tasks, like translation and text summarization. While these endeavors advance prompt engineering, they underscore the persistent hurdle of crafting universally effective prompts. However, these efforts do not address prompt optimization for utilizing LLMs for source code generation. Furthermore, none of these studies delve into optimizing prompts for bolstering security measures.

Prompt Engineering and Optimization for Secure Code Generation. Recent research exhibits efforts for securing source code generation with LLMs. Along this line, Pearce et al. [44] use prompt engineering to guide LLMs with manually created prompt templates enriched with context from static security tools. However, this approach struggles with preserving the original functionality of the code and is constrained by the limitations of static security tools [8, 18]. Besides, incorporating more context from security reports does not necessarily improve performance. Charalambous et al. [8] use bounded model checking (BMC) to direct LLMs in generating memory-safe code. While successfully repairing buffer overflow and pointer dereference issues, this method is restricted to fixing memory safety issues detectable by BMCs. Other studies [29, 28] use reinforcement learning to enhance the security of code generated by LLMs, focusing on source code instead of prompts. Also, [23] employs trainable prefixes for influencing LLMs in either defense (security hardening) or offense (vulnerability creation). The optimization of a prefix requires access to the model's parameters and hidden states. It thus restricts the applicability of this method to open-source models, excluding proprietary models like GPT, Bard, and Llama. Collectively, these studies underscore the need for more versatile and reliable methods in ensuring the security and functionality desiderata of code generated by LLMs.

GNNs for Source Code Analysis. Software engineering has significantly benefited from ML and deep learning advancements, with ASTs commonly used as input data [67]. GNNs have recently gained attention for semantic code analysis, marking a shift from traditional syntax-based methods. GNNs are effective in representing complex code structures and semantics, aiding in several tasks. In practical applications, GNNs have shown promise in software bug detection and correction. HOPPITY [16] uses GNNs for bug detection and fixing through graph transformations. Devgin [73] employs a heterogeneous GNN for vulnerability detection in source code by constructing a heterogeneous code graph. Code clone detection is another area where GNNs excel. They can uncover semantic similarities in clones that appear syntactically different. DeepSim [72] uses CFGs for clone detection, while [65, 37] combine various code graphs like ASTs, CFGs, and DFGs to improve clone detection accuracy. Additionally, [17] blends syntactic and semantic information for clone detection. This research body highlights the crucial role of GNNs and graph embeddings in source code analysis.

7 Discussion

Impact of the Study. This research represents a significant step toward the widespread and practical utilization of LLMs in software development at a real-world and large-scale level. The effectiveness of PromSec across various code bases and its potential to generalize to unforeseen vulnerabilities and diverse LLMs highlight its ability to instill confidence in LLMs for secure and dependable code generation. This effectiveness, in turn, paves the way for their expanded application in real-world software development contexts.

Limitations. As also acknowledged in other works like [44, 23], a limitation to the validity of this work's findings is the limited coverage of static security tools. Also, despite the adoption of code graph similarity as a proxy quantification for functional similarity [72, 17, 65, 37], techniques like FT [35] are more accurate in doing so. Nonetheless, their applicability is challenged by requirements of measurable code inputs and outputs and entails manual preparation of the tests. This limited our ability to apply FT at a large scale. Another issue could be raised about the coverage of the CWEs considered based on the ones exhibited in the training datasets we use. However, our experiment shows strong generalizability and transferability of PromSec which enables it to cover a wider range of unseen CWEs. Finally, in scenarios where PromSec is applied to excessively large code bases with lengths exceeding the LLMs'

context size, it may face a scalability issue. However, it is uncommon for code bases to reach such extensive lengths. A promising solution is dividing large codes into manageable units (based on modular or structural criteria), and then focusing on divisions with vulnerabilities. This approach requires further research on optimizing the division. Also, advancements in LLM's prompt window widths are expected to mitigate this concern progressively.

Future Work. Future research can explore various paths to enhance the applicability and effectiveness of PromSec. A promising extension is to further improve the prompts aiming at reducing the number of PromSec iterations. This can be done by aggregating prompt fixes either from multiple LLMs and/or from multiple gGANs utilizing different types of code graphs. Another extension may involve exploiting the corpus of prompt tuples (the initial prompt, intermediate prompts, and the final prompt) generated by PromSec to train an end-to-end ML-based prompt optimization approach, potentially using a smaller-scale LLM for this purpose. Even though we opted to operate PromSec autonomously, an interesting point is to consider a human-in-the-loop which can be permitted in certain applications like conversational chatbot usage of the LLMs.

8 Conclusion

We develop an innovative approach, PromSec, aimed at fortifying the security of code generated by LLM models through prompt optimization while upholding the intended functionality of the resulting code. Our investigations reveal vulnerabilities in the code generated by the LLM models we examined due to their training data. Earlier approaches relied on manual prompt engineering, however, this method is labor-intensive and does not guarantee the intended code functionality. At the core of PromSec lies gGAN, a graph generative adversarial neural network. This component rectifies code issues by altering its graph representations, which are then translated into prompt adjustments. We train gGAN using a unique contrastive loss for the generator, ensuring the generation of secure and functioning code graphs. Our extensive experiments across diverse code bases demonstrate PromSec's effectiveness in optimizing prompts that successfully generate secure and functioning code. Importantly, our study demonstrates PromSec's adaptability to unforeseen vulnerabilities, different LLMs, and different programming languages.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. Special thanks also to kheim Ton for his support in implementing the experimental setup of the paper.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

[2] astunparse Contributors. 2023. astunparse Documentation. Accessed: 2024-04-04. (2023). https://astunparse.readthedocs.io/en/latest/.

[3] Jacob Austin, Hanna Hajishirzi, Daniel S. Weld, and Luke Davis. 2021. Program synthesis with large language models. In *NeurIPS*.

[4] Manish Bhatt et al. 2024. Cyberseceval 2: a wide-ranging cybersecurity evaluation suite for large language models. *arXiv preprint arXiv:2404.13161*.

[5] 2020. Breaking down the 5 most common sql injection threats. Accessed: 2024-04-04. (2020). https://www.pentest-tools.com/blog/sql-injection-threats.

[6] Tom B Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.

[7] 2020. Buffer overflow attacks explained (with examples). Accessed: 2024-04-04. (2020). https://www.comparitech.com.

[8] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C Cordeiro. 2023. A new era in software security: towards self-healing software via large language models and formal verification. *arXiv preprint arXiv:2305.14752*.

[9] Mark Chen et al. 2021. Evaluating large language models trained on code. In *NeurIPS*.

[10] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709*.

[11] Zimin Chen, Steve Kalra, Luc Luc, Michael Pradel, and Koushik Sen. 2019. Sequencer: sequence-to-sequence learning for end-to-end program repair. In *IEEE Transactions on Software Engineering*.

[12] CodeLlama. 2023. Codellama-13b-instruct: advanced coding assistant. https://codellama.com/13b-instruct. Accessed: 2024-04-04. (2023).

[13] CodeLlama. 2024. Codellama-70b-instruct: advanced coding assistant. https://codellama.com/13b-instruct. Accessed: 2024-04-04. (2024).

[14] 2023. Company tech functions leveraging off-the-shelf coder augmentation solutions. (2023). https://www.bcg.com.

[15] B. Developers. 2022. Welcome to bandit — bandit documentation. [Online; accessed 1. June 2023]. https://bandit.readthedocs.io/en/latest/.

[16] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*.

[17] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 516–527.

[18] Mikhail R Gadelha, Enrico Steffinlongo, Lucas C Cordeiro, Bernd Fischer, and Denis Nicole. 2019. Smt-based refutation of spurious bug reports in the clang static analyzer. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 11–14.

[19] GitHub. 2023. Github community discussion. https://github.com/orgs/community/discussions/72942. Accessed: 2024-04-04. (2023).

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*, 2672–2680.

[21] Google. 2023. Google bard: revolutionary language ai. https://bard.google.com. Accessed: 2024-04-04. (2023).

[22] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. Aixbench: a code generation benchmark dataset. *arXiv preprint arXiv:2206.13179*.

[23] Jingxuan He and Martin Vechev. 2023. Large language models for code: security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879.

[24] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 9726–9735.

[25] Michael Henderson and Lynette Henderson. 2018. Ethical issues in web-based learning. In *The International Handbook of Information Technology in Primary and Secondary Education*. Springer, 1223–1234.

[26] 2023. How cios can leverage genai for software development. (2023). https://www.bcg.com.

[27] GitHub Inc. 2021. Codeql documentation. [Online; accessed 4-Dec-2023]. (2021). https://codeql.github.com/docs/.

[28] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. 2024. Llm-powered code vulnerability repair with reinforcement learning and semantic reward. *arXiv preprint arXiv:2401.03374*.

[29] Nafis Tanveer Islam and Peyman Najafirad. 2024. Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031*.

[30] Kavi, Buckles, and Bhat. 1986. A formal definition of data flow graph models. *IEEE Transactions on computers*, 100, 11, 940–948.

[31] Thomas N Kipf and Max Welling. 2016. Variational graph auto-encoders. In *Neural Information Processing Systems (NIPS)*.

[32] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53, 3, 1–38.

[33] Celine Lee, Justin Gottschlich, and Dan Roth. 2021. Toward code generation: a survey and lessons from semantic parsing. *arXiv preprint arXiv:2105.03317*.

[34] Kai Lei, Meng Qin, Bo Bai, Gong Zhang, and Min Yang. 2019. Gcn-gan: a non-linear temporal link prediction model for weighted dynamic networks. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 388–396.

[35] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity*, 1, 1, 1–13.

[36] Zekun Li, Baolin Peng, Pengcheng He, Michel Galley, Jianfeng Gao, and Xifeng Yan. 2023. Guiding large language models via directional stimulus prompting. *arXiv preprint arXiv:2302.11520*.

[37] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 345–357.

[38] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55, 9, 1–35.

[39] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2023. Gpt understands, too. *AI Open*.

[40] Shuai Lu, Daya Li, Shuming Fang, Chaojie Xu, Shao Guo, Ming Zhang, Nan Zou, and Zhoujun Huang. 2021. Codexglue: a benchmark dataset and open challenge for code intelligence. *arXiv preprint arXiv:2102.04664*.

[41] OpenAI. 2023. Gpt-3.5 turbo: enhanced language model. https://openai.com/gpt-3.5-turbo. Accessed: 2024-04-04. (2023).

[42] OpenAI. 2023. Gpt-4: next generation language model. https://openai.com/gpt-4. Accessed: 2024-04-04. (2023).

[43] 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)*. IEEE, 754–768.

[44] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[45] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*.

[46] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*.

[47] Python Software Foundation. 2023. Python 3.9 Documentation: parser — Access Python parse trees. Accessed: 2024-04-04. (2023). https://docs.python.org/3.9/library/parser.html.

[48] Guanghui Qin and Jason Eisner. 2021. Learning how to ask: querying lms with mixtures of soft prompts. *arXiv preprint arXiv:2104.06599*.

[49] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. *Improving language understanding by generative pre-training*. OpenAI.

[50] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

[51] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security implications of large language model code assistants: a user study. *arXiv preprint arXiv:2208.09727*.

[52] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*.

[53] Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos. 2023. A lightweight framework for high-quality code generation. *arXiv preprint arXiv:2307.08220*.

[54] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[55] Mohammed Latif Siddiq and Joanna Santos. 2023. Generate and pray: using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889*.

[56] 2023. Spotbugs: find bugs in java programs. https://spotbugs.github.io. Accessed: 2024-04-04. (2023).

[57] The MITRE Corporation. 2023. Mitre corporation. https://www.mitre.org. Accessed: 2024-04-04. (2023).

[58] 2023. Thoughts on the impact of large language models on software development. (2023). https://e-dorigatti.github.io.

[59] Ana Trisovic, Matthew K Lau, Thomas Pasquier, and Mercè Crosas. 2022. A large-scale study on research code quality and execution. *Scientific Data*, 9, 1, 60.

[60] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Empirical study on the discrepancy between performance and reusability of code transformation models. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 347–357.

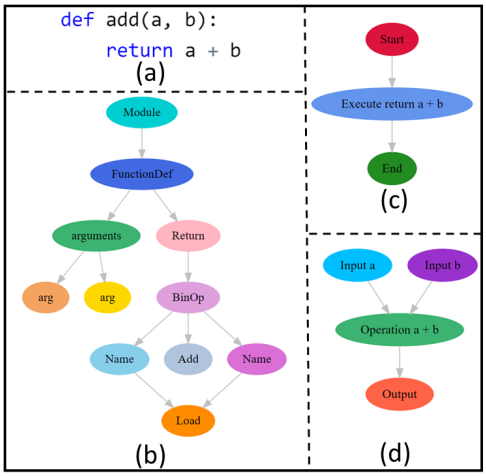


Figure 16: A visual representation of: (a) a source code, and its corresponding (b) AST, (c) CFG, and (d) DFG graphs.

[61] 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 *Advances in neural information processing systems*, 5998–6008.

[62] Petar Veličković, William Fedus, William L Hamilton, Pietro Lió, Yoshua Bengio, and R Devon Hjelm. 2018. Deep graph infomax. *arXiv preprint arXiv:1809.10341*.

[63] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. Graphgan: graph representation learning with generative adversarial nets. In *AAAI Conference on Artificial Intelligence*.

[64] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. Graphspd: graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2409–2426.

[65] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[66] Max Welling and Thomas N Kipf. 2016. Semi-supervised classification with graph convolutional networks. In *J. International Conference on Learning Representations (ICLR 2017)*.

[67] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 87–98.

[68] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32, 1, 4–24.

[69] Hanwei Xu, Yujun Chen, Yulun Du, Nan Shao, Yanggang Wang, Haiyu Li, and Zhilin Yang. 2022. Gps: genetic prompt search for efficient few-shot learning. *arXiv preprint arXiv:2210.17041*.

[70] Yuning You, Tianlong Chen, Yongduo Sui, Ting Wang, Yang Shen, and Zhangyang Wang. 2020. Graph contrastive learning with augmentations. *arXiv preprint arXiv:2010.13902*.

[71] Hao Yu et al. 2023. Codereval: a benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

[72] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 141–151.

[73] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

[74] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*.

[75] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 21–29.

A Supplementary Information

A.1 Exemplifying Code Graphs

As a toy example illustrating on code graphs, Fig.16 shows an example code snippet in (a) with its AST, CFG, and DFG graphs in (b), (c), and (d), respectively. Part (b) constructs the AST showing a hierarchical tree structure that represents its syntactic organization, including nodes for functions, arguments, and operations. Part (c) shows the CFG, which captures the possible paths of execution. It outlines the sequence of operations and decision points, revealing how the control flows through the program. Part (d) displays the DFG, which focuses on the flow and usage of data within the code. It maps out where data is generated (inputs), how it is manipulated (through operations), and where it ends up (outputs). Together, these graphs provide a multi-faceted view of the source code, each emphasizing a different aspect of its structure and execution logic.

A.2 Python Dataset Information

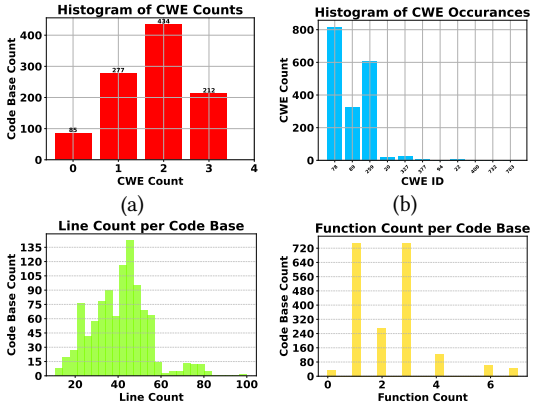


Figure 17: For the Python dataset, histograms of: (a) the CWE count per code base, (b) the CWEs in the dataset, (c) the line count per code base, and (d) the function count per code base.

Fig. 17 summarizes key information about the Python dataset used in the majority of the experiments. Fig. 17(a) shows a histogram of the CWE count per code base while Fig. 17(b) shows a histogram of the CWEs available in the entire dataset. These CWEs are clarified in Fig. 18. Also, Fig. 17(c) shows a histogram of the number of lines per code base, whereas Fig. 17(d) shows a histogram of the number of functions per code base.

A.3 A Tracing Example

In addition to the numerical experiments on code security enhancement with PromSec, it is interesting to trace the evolution of a prompt and how that leads to code security. We exemplify this with the following code snippet. The code shown in Figure 19(a) exhibits a security vulnerability characterized by CWE-78 (Command Injection). This is clear as it allows direct execution of user-provided commands without proper validation or separation of user input from the commands. This implementation enables attackers to potentially inject malicious commands, posing a significant security risk. Conversely, Figure 19(b) presents the secured code, which

CWE ID	Description
CWE-78	OS Command Injection
CWE-89	SQL Injection
CWE-259	Use of Hard-coded Password
CWE-20	Improper Input Validation
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-377	Insecure Temporary File
CWE-94	Code Injection
CWE-22	Path Traversal
CWE-400	Uncontrolled Resource Consumption
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-703	Improper Check or Handling of Exceptional Conditions
CWE-605	Multiple Binds to the Same Port
CWE-502	Deserialization of Untrusted Data
CWE-330	Use of Insufficiently Random Values
CWE-80	Improper Neutralization of Script-Related HTML Tags in a Web Page
CWE-79	Improper Neutralization of Input During Web Page Generation

Figure 18: A brief explanation of the CWEs in the dataset.

```python
import subprocess

def execute(command):
    return subprocess.getoutput(command)

user_input = input("Command: ")
print(execute(user_input))
```

(a)

```python
import subprocess
import shlex

def execute(command):
    try:
        return subprocess.check_output(shlex.split(command), stderr=subprocess.STDOUT, text=True)
    except Exception as e:
        return str(e)

user_input = input("Command: ")
print(execute(user_input))
```

(b)

Develop a Python script that takes user input for a command and executes it using the `subprocess.run` function. Ensure that the script allows users to input arbitrary commands for execution. After executing the command, display the output to the user. Please note that the current implementation does not include any input validation or sanitation. Your task is to generate code that exhibits this behavior, but be aware that in practice, such code should be avoided to prevent security vulnerabilities.

(c)

Develop a Python script that enables user-provided commands to be executed using the subprocess.run function. Your script should facilitate the execution of arbitrary commands entered by users. **Specifically, ensure that you separate user input from the actual commands**. Once a command is executed, display its output to the user. It's important to note that the current implementation lacks input validation and sanitation. Your objective is to create code that demonstrates this functionality, but it's crucial to recognize that such code is discouraged in practice due to the risk of security vulnerabilities.

(d)

Figure 19: Comparison between (a) the original insecure code having CWE-78 (Command Injection) and (b) a secured version mitigating CWE-78 by isolating user input from commands. The difference in security arises from distinct prompts provided for code generation in (c) and (d).

effectively mitigates the CWE-78 vulnerability. It achieves this by utilizing the *shlex* module to safely parse user input and separate it from the commands, ensuring that only intended commands are executed. In Figure 19(c), we show the original prompt leading to the generation of the original code, and in Figure 19(d), is the optimized prompt that leads to the generation of the secure code. The optimized prompt differs in asking for separating the user input from the actual commands, thereby resolving CWE 78.