UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

SÉCURITÉ DES CHATBOTS AI - ANALYSE AUTOMATISÉE DE LA SÉCURITÉ DU CODE AVEC CHATGPT

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

PAR

REZIKA BOUZID

NOVEMBRE 2024

Ce mémoire a été évalué par un jury composé des personnes suivantes :

Dr. Alan Davoust ..... Président du jury

Dr. Zakaria Abou El Houda ..... Membre du jury

Dr. Raphaël Khoury ..... Directeur de recherche

Mémoire accepté le : 18 Avril 2025

Dédicace

À mon mari, pour ta patience et ton soutien inestimables. Ton encouragement constant a été ma plus grande force. À mes enfants, si gentils et adorables, vous êtes ma source d'inspiration, de joie et de détermination. À ma mère, pour ton amour inconditionnel, tes sacrifices et ton soutien précieux tout au long de mon parcours. Et à toutes les personnes guidées par la soif d'apprendre et de progresser.

Remerciements

Je tiens à exprimer ma gratitude infinie à Allah, le Tout-Puissant, pour Sa guidance, Sa miséricorde et les innombrables bénédictions qui m'ont soutenu tout au long de ce parcours. Sans son désir, rien de tout cela n'aurait pu se concrétiser.

Je souhaite également exprimer ma profonde reconnaissance à mon directeur de recherche, le Professeur Raphaël Khoury, pour ses conseils précieux, son encadrement de qualité, son expertise et son soutien constant tout au long de ce travail de recherche. Je remercie chaleureusement les membres du jury pour le temps qu'ils ont consacré à l'évaluation de ce travail. Ma reconnaissance va également à ma famille, pour leur amour inconditionnel, leur patience et leur support permanent dans toutes les étapes de ce parcours. Enfin, ces remerciements s'adressent à tous ceux qui, par leurs contributions, leur aide, leur inspiration ou leur bienveillance, ont rendu cette aventure possible.

Table des matières

Liste des figures **iv**

Liste des tableaux **v**

Liste des abréviations, sigles et acronymes **vii**

Résumé **viii**

1 Introduction **1**

1.1 Contexte 1

1.2 Problématique 1

1.3 Objectifs 2

1.4 Contributions 3

2 Etat de l'art **5**

2.1 L'intelligence artificielle 6

2.1.1 L'apprentissage automatique 6

2.1.2 L'apprentissage profond 8

2.1.3 IA générative 8

2.1.4 Grands modèles de langage 8

2.1.5 Transformers 9

2.1.6 Chatbot basé sur l'IA 10

2.1.7 ChatGPT 11

2.1.8 Ingénierie de prompt 12

2.1.9 Tokens dans le contexte des LLMs 12

2.2 La sécurité logicielle 13

2.2.1 Définition 13

2.2.2 Vulnérabilités 14
2.2.3 Vulnérabilités et Expositions Courantes 14
2.2.4 Énumération des Faiblesses Courantes 15
2.2.5 Base de Données Nationale des Vulnérabilités 15
2.3 Revue de la littérature 16
2.3.1 Questions de recherche 17
2.3.2 Catégorisation des études sélectionnées 18
2.4 Travaux connexes 21

3 Méthodologie **26**
3.1 Collecte de données 27
3.1.1 Description de la base de données 27
3.1.2 Méthode d'extraction 28
3.1.3 Limitations du jeu de données 31
3.1.4 Prétraitement des données extraites 32
3.2 Interaction avec le LLM 33
3.3 Désigne du prompt 34
3.3.1 La structure 34
3.3.2 Le contenu 34
3.4 Expérimentation 35
3.4.1 Scénario d'évaluation 36
3.4.2 Métriques d'évaluation 39

4 Analyse des résultats **43**
4.1 Détection de vulnérabilités 44
4.1.1 Classification binaire 44
4.1.2 Classification des codes bénins 44
4.2 Classification des vulnérabilité selon le type CWE 45
4.2.1 Performances avant et après la date d'entraînement 45
4.2.2 Performances en fonction des CWE les plus récurrentes 47
4.3 Localisation des lignes vulnérables 48
4.4 Correction automatique de vulnérabilités 52
4.4.1 Correctif pour un code vulnérable 52
4.4.2 Correctif pour un code bénin 54
4.5 Re-vérification de l'existence de vulnérabilités 54

4.6 Performances de ChatGPT comparées à celles d'autres LLMs 56
4.6.1 Classification binaire 57
4.6.2 Classification des vulnérabilités selon le type CWE 58
4.6.3 Localisation des lignes vulnérables 65
4.6.4 Correction automatique de vulnérabilités 66
4.6.5 Re-vérification de l'existence de vulnérabilités 71
4.7 Leçons tirées de l'utilisation de trois LLMs comme réviseurs de la sécurité du code 73
4.7.1 Limite des tokens : un obstacle à l'analyse de programmes complets 73
4.7.2 CVEs traitées par les trois LLMs diffèrent 73
4.7.3 Divergence dans la capacité des modèles 74
4.7.4 Incomplétude des réponses générées par les LLMs 74
4.7.5 Incohérence des réponses : un obstacle à l'évaluation automatisée des LLMs 74
4.7.6 Anomalies dans les réponses des LLMs 75

5 Conclusion **76**

Bibliographie **79**

Liste des figures

2.1 Les différents niveaux de l'IA 6
2.2 Architecture des Transformers, tirée de l'article original [46] 10
2.3 Calculatrice de tokens pour les modèles GPTs -21/07/2024 [1] 13
2.4 Détails de la Vulnérabilité CVE-2023-12345 15

3.1 Architecture de notre système 26
3.2 Le schéma de la base de données CVEFixes extrait de l'article [23] 28
3.3 La requête SQL utilisée pour extraire les données à partir de la base de données CVEFixes 30
3.4 Extrait du fichier d'entrée 31
3.5 Les différents prompts employés pour interagir avec ChatGPT 35

4.1 Performances de ChatGPT pour la classification des vulnérabilités selon le type CWE avant et après la date d'entraînement. 46
4.2 L'exactitude de ChatGPT dans la classification des CWE récurrentes . . 48
4.3 Exemples de formats variés de réponses de ChatGPT pour les lignes vulnérables 50
4.4 Comparaison des performances de ChatGPT a celles de Llama et DeepSeek dans la classification des vulnérabilités selon le type CWE pour les langages C, C++ et Java. 62
4.5 Comparaison des performances de ChatGPT avec celles de Llama et DeepSeek dans la classification des vulnérabilités selon le type CWE pour les langages PHP et Python 63
4.6 Comparaison de l'exactitude de ChatGPT dans la classification des CWE récurrentes à celles de Llama et DeepSeek 64

Liste des tableaux

2.1 Comparaison des études utilisant GPT-3.5 Turbo pour l'analyse de la sécurité du code, en fonction des contextes et des tâches de sécurité évaluées (I : Identification, S : Sévérité, T : Type CWE, L : Lignes Vulnérables, C : Correction, Codes Complets(CC), Fonctions (F), Vulnérables (V), Bénins (B), Vulnérables et Bénins(VB)) 24

3.1 Répartition des codes testés par langages de programmation 33
3.2 Descriptions des différents rôles dans le message d'entrée 34

4.1 Classification binaire par langage de programmation avant et après septembre 2021. 44
4.2 Taux FPR par langages de programmation avant et après septembre 2021. 45
4.3 Liste des CWE récurrentes analysées 49
4.4 Mesure des indices IoU et IFA pour les codes vulnérables (V) et bénins (B) avant et après septembre 2021. 51
4.5 Mesure des indices IoU et IFA pour les codes vulnérables correctement dentifiés ainsi que leurs types CWE. 53
4.6 Performances de ChatGPT dans la detection de lignes vulnerables au second tour de vérification 55
4.7 Les modèles de langage étudiés 56
4.8 Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans la classification binaire 57
4.9 Comparaison des performances de ChatGPT avec celles de Llama et de DeepSeek dans la classification binaire par langage de programmation . . 58
4.10 Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans l'attribution du type CWE pour les codes vulnérables 59

4.11 Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans l'attribution du type CWE pour les codes bénins . . 60
4.12 Comparaison des valeurs de l'IFA et IoU pour les 3 modèles sur les codes vulnérables et bénins, par langage de programmation. 66
4.13 Résultats IoU et IFA par langage pour chaque modèle, permettant d'identifier les correctifs à évaluer 67
4.14 Résultats de la re-vérification des vulnérabilités correspondant au type CWE indiqué pour chaque LLM 71
4.15 Résultats IoU et IFA pour chaque modèle permettant d'évaluer les nouvelles lignes predites après la re-vérification 72

Liste des abréviations, sigles et acronymes

AI Artificial Intelligence

API Application Programming Interface

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

GAN Generative Adversarial Network

GPT Generative Pre-trained Transformer

LLM Large Language Model

LSTM Long Short-Term Memory

MDP Markov Decision Process

NIST National Institute of Standards and Technology

NLP Natural Language Processing

NVD National Vulnerability Database

RNN Recurrent Neural Network Architecture

Résumé

L'automatisation de l'analyse de la sécurité du code constitue un défi majeur, surtout avec l'émergence des grands modèles de langage (LLMs), fruits des avancées significatives dans le domaine de l'intelligence artificielle (IA), comme le chatbot ChatGPT.

Le présent mémoire vise à évaluer l'efficacité de ChatGPT dans l'identification des vulnérabilités, la classification de chaque vulnérabilité selon son type CWE, la localisation des lignes vulnérables et la suggestion de correctifs appropriés dans des codes sources complets issus de plusieurs langages de programmation tirés de scénarios CVEs réels, issus de projets logiciels véritables. Ce processus simule une interaction instantanée sous forme de dialogue, permettant d'analyser l'ensemble du code pour toutes les tâches en une seule fois, plutôt que de tester chaque tâche indépendamment, ce qui permet de maintenir le contexte du code visé. Nous mettons en lumière ses points forts ainsi que ses limites, tant avant qu'après la fin de son entraînement, et le comparons à deux autres LLMs, DeepSeek et Llama.

Les performances de ChatGPT varient selon les tâches et langages, s'améliorant après septembre 2021 pour certaines, mais se dégradant pour d'autres. Comparé aux autres LLMs, ChatGPT a été surpassé dans certaines tâches de sécurité et langages, tandis que dans d'autres cas, il les a surpassé. Toutefois, les trois modèles se retrouvent sur un pied d'égalité en ce qui concerne la génération de faux positifs, et aucun d'entre eux n'a réussi à produire un bilan positif. De plus, le deuxième tour de vérification des vulnérabilités n'a pas été concluant pour les trois modèles en ce qui concerne la localisation des lignes vulnérables, à l'exception de Llama, qui a surpassé ChatGPT et DeepSeek dans la confirmation de l'existence des vulnérabilités dans tous les cas.

Les LLMs ne sont pas encore fiables pour l'analyse de sécurité des programmes, car leurs performances sur du code réel restent moins bonnes que des résultats optimistes des recherches précédentes.

Mots-clés : *ChatGPT, LLM, vulnérabilités, sécurité du code.*

Abstract

Automating code security analysis remains a major challenge, especially with the emergence of large language models (LLMs), which are the result of significant advancements in artificial intelligence (AI), such as the ChatGPT chatbot.

This thesis aims to evaluate the effectiveness of ChatGPT in identifying vulnerabilities, classifying each vulnerability by its CWE type, locating vulnerable lines, and suggesting appropriate fixes within complete source codes written in various programming languages derived from real CVEs scenarios and concrete software projects. This process simulates an instant interaction in the form of a dialogue, enabling the analysis of entire codebases for all tasks simultaneously rather than testing each task independently, thereby preserving the targeted code's context. We highlight its strengths and limitations, both before and after the end of its training, and compare it to two other LLMs, DeepSeek and Llama.

ChatGPT's performance varies depending on tasks and languages, improving after September 2021 for some but deteriorating for others. Compared to other LLMs, ChatGPT was outperformed in certain security tasks and languages, while in other cases, it outperformed its counterparts. However, all three models are on equal footing regarding the generation of false positives, and none succeeded in producing a fully accurate assessment. Additionally, the second round of vulnerability verification was inconclusive for all three models regarding the localization of vulnerable lines, except for Llama, which outperformed ChatGPT and DeepSeek in confirming the existence of vulnerabilities in all cases.

Overall, LLMs are not yet reliable for program security analysis, as their performance on real-world code falls short of the optimistic results reported in prior research.

Keywords : *ChatGPT, LLM, vulnerabilities, code security.*

Chapitre 1

Introduction

1.1 Contexte

ChatGPT, développé par OpenAI [2], est un Grand Modèle de Langage (LLM) qui a été minutieusement entraîné sur des ensembles de données volumineux pour prédire les séquences de mots dans les phrases avec une précision exceptionnelle. En utilisant des techniques d'apprentissage profond et l'architecture Transformer [46], ChatGPT ouvre des possibilités inédites pour des applications au-delà de l'interprétation linguistique traditionnelle. Depuis son lancement en 2022, ChatGPT a été rapidement adopté dans divers secteurs et applications, transformant la manière dont les tâches quotidiennes sont accomplies.

Une application suscitant un grand intérêt est la programmation logicielle. Les développeurs comptent de plus en plus sur ChatGPT pour générer du code et accomplir des tâches liées à la programmation sécurisée. Cependant, ils se demandent souvent si le code produit est réellement sécurisé.

1.2 Problématique

Étant donné l'intérêt croissant pour cette technologie, qui contribue de manière significative à l'amélioration de la productivité, il est essentiel de s'assurer que l'utilisation de ce chatbot IA soit digne de confiance et présente de faibles risques. Par conséquent, il est impératif de vérifier la sécurité et la fiabilité des résultats produits par le LLM à la suite de son travail d'analyse.

Plus particulièrement, il est important d'explorer dans quelle mesure ChatGPT-3.5 est capable de détecter, localiser, classer et corriger les vulnérabilités dans le même code, ainsi que d'évaluer comment ces capacités évoluent avant et après sa date d'entraînement, fixée à septembre 2021. De plus, la mise en œuvre d'un second tour pour vérifier l'existence réelle de vulnérabilités dans les cas de faux négatifs ou de faux positifs améliore-t-elle les résultats ? Les corrections proposées par le modèle ajoutent-elles de la complexité au code initial ? Enfin, quelles sont ses capacités en analyse de sécurité du code comparées à celles d'autres LLMs ?

Compte tenu de la nouveauté du sujet, malgré les efforts considérables dans ce sens où plusieurs techniques ont déjà été adaptées pour évaluer ChatGPT pour la génération du code, il existe un manque notable dans la littérature en ce qui concerne l'utilisation de ce modèle pour l'analyse de sécurité du code. La plupart des recherches se concentrent sur les capacités des LLM à générer du code sémantiquement et syntaxiquement correct ou à assister les développeurs dans des tâches de programmation générale. Cependant, peu d'études approfondissent l'efficacité de ce modèle dans la détection et la correction des vulnérabilités de sécurité.

1.3 Objectifs

Ce projet vise à évaluer l'efficacité de ChatGPT en matière de sécurité du code et à déterminer si le LLM peut, de façon autonome et sans indications spécifiques, identifier les vulnérabilités, classer chaque vulnérabilité selon son type, localiser les lignes vulnérables et suggérer des correctifs appropriés. Pour ce faire, nous lui fournirons des codes sources complets issus de scénarios CVEs réels tirés de véritables projets logiciels et lui demanderons d'effectuer les tâches de sécurité précédentes dans un ordre précis via des prompts spécialement conçus pour chaque tâche. Cette approche permet de simuler une interaction instantanée sous forme de dialogue, permettant d'analyser le même code pour toutes les tâches en une seule fois, plutôt que de tester la capacité de chaque tâche indépendamment ce qui maintient le contexte du code visé. Cela permet de produire un bilan complet, positif si le modèle réussit toutes les tâches d'analyse pour un code vulnérable, ou un bilan négatif s'il classe correctement les codes bénins. Cette méthode est plus réaliste que les "codes jouets" souvent utilisés dans les premières études sur la sécurité du code généré par ChatGPT, car elle reflète mieux les scénarios réels où les vulnérabilités ne sont pas préalablement identifiées. Elle exige donc que le LLM

mobilise ses connaissances acquises durant la phase d'entraînement. De plus, elle offre la possibilité de scanner l'intégralité du code et de vérifier entièrement sa sécurité.

La motivation de cette évaluation est double : d'une part, déterminer la fiabilité des LLMs en tant qu'outils d'analyse de sécurité ; d'autre part, fournir des éléments de réflexion pour étudier la faisabilité de leur intégration future dans les environnements de développement, afin d'offrir aux développeurs des alertes et des recommandations de sécurité pertinentes.

1.4 Contributions

Pour ces raisons, ce travail apporte les contributions suivantes :

- Nous proposons une revue complète de la littérature sur les techniques utilisées pour évaluer les performances de ChatGPT dans le domaine de la sécurité du code. Un article à cet effet est présentement soumis à un journal académique.
- Nous appliquons une méthode de simulation d'interaction instantanée avec le LLM via son API pour analyser un code complet en une seule session, plutôt que de tester les capacités du modèle pour chaque tâche de sécurité individuellement. En conservant le même contexte tout en respectant la limite de tokens. Ainsi, elle permet une évaluation plus cohérente et exhaustive de la sécurité du code dans son intégralité.
- Nous testons à la fois du code vulnérable et du code bénin pour capturer le comportement du LLM dans les deux cas. En incluant ces deux types de code, nous pouvons observer comment le modèle réagit et s'adapte aux différentes situations afin de détecter les faux positifs éventuels. Cela nous permet de mieux comprendre ses capacités à distinguer entre les codes sûrs et ceux qui présentent des risques de sécurité et évaluer son efficacité et/ou la complexité des solutions proposées.
- Lors de l'analyse initiale d'un code vulnérable par le LLM, si des faux négatifs (le modèle ne détecte pas une vulnérabilité qui existe réellement) ou des faux positifs (le modèle trouve une vulnérabilité du mauvais type CWE) sont détectés dans ses réponses, nous effectuons un deuxième tour pour vérifier l'existence réelle de vulnérabilités. Cette re-vérification est possible grâce à la sauvegarde des réponses du LLM pendant l'interaction, permettant une analyse immédiate des résultats. Pour cette deuxième analyse, nous utilisons un prompt spécifique contenant des indications supplémentaires.

– Nous examinons, à travers plusieurs langages de programmation, dans quelle mesure GPT-3.5 Turbo peut détecter, localiser, classer et corriger les vulnérabilités dans le code. Nous analysons ses performances et son comportement, tant sur des vulnérabilités qu'il a déjà rencontrées lors de son entraînement, achevé en septembre 2021, que sur celles qu'il n'a pas encore vues. De plus, nous comparons ces performances à celles de deux autres LLMs, DeepSeek et Llama, apportant ainsi une contribution notable à la cybersécurité dans le développement logiciel.

Avant septembre 2021, ChatGPT génère de nombreux faux positifs et échoue dans 94,13% des cas pour les codes bénins. Après cette date, ses performances s'améliorent pour certains langages. De plus, aucun des trois modèles, ChatGPT, DeepSeek ou Llama, n'était capable de fournir un bilan positif, c'est-à-dire qu'aucun code analysé n'a obtenu une réponse correcte pour l'ensemble des tâches simultanément. Pour les codes bénins dans les données communes, ChatGPT a généré 16 bilans négatifs, contre 9 pour Llama et 30 pour DeepSeek, tout en affichant un taux élevé de faux positifs. De plus, le deuxième tour de vérification des vulnérabilités s'est révélé infructueux pour les trois modèles en matière de localisation des lignes vulnérables, à l'exception de Llama, qui a surpassé ChatGPT et DeepSeek dans la confirmation de l'existence des vulnérabilités du type CWE spécifié pour tous les cas. Par ailleurs, les correctifs proposés par ChatGPT ne résolvent pas les vulnérabilités et complexifient souvent le code initial en modifiant des parties non vulnérables, un constat également observé pour les deux autres LLMs.

Ce mémoire est structuré pour offrir une vue d'ensemble complète et cohérente de notre étude sur les performances de ChatGPT en matière de détection et de correction des vulnérabilités dans le code. Il commence par le Chapitre 1, qui présente le contexte général de l'étude, la problématique et les objectifs visés. Le Chapitre 2 est dédié à l'état de l'art et inclut les fondements des concepts techniques, une revue de la littérature et une analyse des travaux connexes. Ensuite, le Chapitre 3 détaille la méthodologie employée, incluant la collecte de données servant de référence, l'interaction avec le LLM via l'API, la conception des prompts et l'expérimentation, avec une description détaillée du scénario d'évaluation et des métriques utilisées. Le Chapitre 4 expose les résultats détaillés de l'évaluation, tandis que le Chapitre 5 conclut le document en discutant des résultats et des perspectives pour les recherches futures.

Chapitre 2

Etat de l'art

La sécurité logicielle, essentielle depuis des décennies, reste cruciale pour garantir l'absence de vulnérabilités dans le code et prévenir les attaques malveillantes. Diverses techniques, telles que l'analyse statique, l'analyse dynamique et les algorithmes d'apprentissage automatique, ont été développées pour détecter des vulnérabilités dans les programmes.

Cependant, récemment, l'arrivée des LLMs, et en particulier de Transformateur Génératif Pré-entraîné (GPT), a bouleversé le travail des développeurs. Ces modèles intelligents sont de plus en plus utilisés pour écrire du code. En conséquence, il est crucial de s'adapter à ce changement et de vérifier la sécurité du code produit par ces modèles et de s'assurer qu'il n'y a aucun risque lors de leur intégration dans les tâches de sécurité liées au développement logiciel.

Ce chapitre mettra l'accent sur plusieurs aspects clés. Tout d'abord, il offrira un aperçu des notions d'intelligence artificielle, notamment des modèles intelligents comme les LLM, et présentera un bref survol des principaux concepts techniques utilisés dans la sécurité logicielle, nécessaires à la compréhension de cette étude. Ensuite, une revue de la littérature se concentrera sur les recherches évaluant ChatGPT dans la génération de code sécurisé et les tâches de sécurité, en résumant et détaillant les méthodes d'évaluation employées. Enfin, notre travail sera situé dans le contexte des recherches existantes en présentant un résumé des travaux connexes.

2.1 L'intelligence artificielle

«L'intelligence artificielle (IA) désigne un ensemble de techniques qui permettent aux ordinateurs de reproduire des comportements humains, tels que l'acquisition de connaissances, la prise de décisions, la détection de motifs et la résolution de problèmes complexes, de manière similaire à celle d'un être humain». [10]. La figure 2.1 reprénsente les différentes sous-catégorie de l'IA [10].

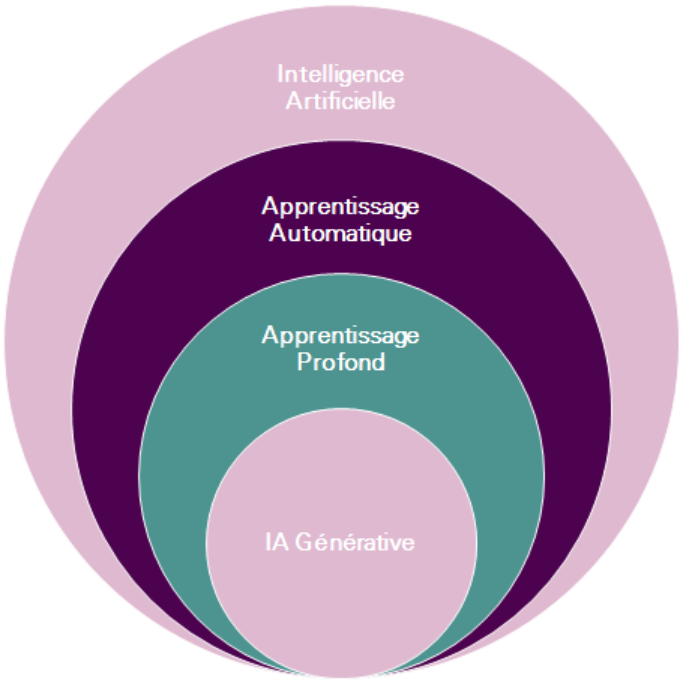

FIGURE 2.1 – Les différents niveaux de l'IA

2.1.1 L'apprentissage automatique

Apprentissage Automatique est une branche de l'IA qui emploie des algorithmes sophistiqués pour repérer des modèles dans de vastes volumes de données, facilitant ainsi l'apprentissage et l'adaptation des machines [10]. Le processus d'apprentissage basé sur les données est rendu possible grâce à ces algorithmes, qui sont classés en trois catégories principales, comme suit :

A. Apprentissage supervisé

L'apprentissage supervisé est une méthode d'entraînement des modèles d'apprentissage automatique où le modèle est formé sur un ensemble de données étiquetées. Chaque exemple de l'ensemble de données d'entraînement est associé à une réponse ou une étiquette correcte. L'objectif est d'apprendre une fonction, souvent implémentée par des algorithmes d'apprentissage automatique tels que les régressions, les arbres de décision ou les réseaux de neurones, qui peut prédire les étiquettes des nouvelles données en se basant sur les exemples appris. Ce processus implique l'ajustement des paramètres du modèle pour minimiser l'erreur entre les prédictions du modèle et les étiquettes réelles[36].

B. Apprentissage non-supervisé

L'apprentissage non supervisé est une méthode d'apprentissage automatique où les algorithmes analysent des données qui ne sont pas accompagnées de labels ou d'informations préalables (données non étiquetées). Ces algorithmes cherchent à identifier des modèles cachés ou à regrouper les données en fonction de similitudes ou de différences sans intervention humaine. En d'autres termes, l'algorithme explore les données pour détecter des structures ou des relations sous-jacentes sans qu'on lui indique explicitement ce qu'il doit rechercher. Cette approche est particulièrement utile pour des tâches comme l'analyse exploratoire des données, où l'on cherche à mieux comprendre des ensembles de données complexes, ou encore pour des applications telles que la segmentation des clients, la recommandation de produits et la reconnaissance d'images, où il est important de détecter des motifs ou des groupes dans les données [14].

C. Apprentissage par renforcement

L'apprentissage par renforcement est une méthode d'apprentissage automatique où un agent apprend à prendre des décisions en interagissant avec un environnement. L'agent reçoit des récompenses ou des pénalités en fonction de ses actions et ajuste ses comportements pour maximiser la récompense cumulée au fil du temps.

Les modèles sophistiqués d'apprentissage par renforcement fonctionnent dans des environnements complexes pour atteindre des objectifs précis, apprenant généralement de manière autonome grâce à un feedback positif ou négatif de l'environnement afin d'optimiser les décisions futures. Ces interactions sont généralement définies par un processus de décision de Markov (MDP) [45].

2.1.2 L'apprentissage profond

«L'apprentissage profond est une branche de l'apprentissage automatique qui utilise des réseaux neuronaux pour analyser les données en profondeur et accomplir des tâches complexes. L'AP se sert de multiples couches de réseaux neuronaux pour extraire des caractéristiques avancées des données, imitant ainsi la manière dont le cerveau humain perçoit et interprète les informations.»[10].

2.1.3 IA générative

L'IA générative est une sous-catégorie de l'intelligence artificielle qui se concentre sur la création de nouveaux contenus ou données qui imitent ou sont inspirés par des données existantes. Contrairement aux IA discriminatives, qui se concentrent sur la classification ou la prédiction à partir de données, les IA génératives ont pour objectif de produire des échantillons de données (comme du texte, des images, ou des vidéos) qui ressemblent à ceux de leur ensemble d'apprentissage. Elles utilisent des techniques telles que les réseaux de neurones génératifs, y compris les réseaux antagonistes génératifs (GAN) et les modèles de transformateurs, pour créer des résultats nouveaux et cohérents en se basant sur des modèles appris à partir de grandes quantités de données [28, 29].

2.1.4 Grands modèles de langage

Les grands modèles de langage, tels que ChatGPT, utilisent des architectures d'apprentissage profond basées sur les réseaux de neurones et les techniques d'IA générative, comme les Transformers, pour exceller dans les tâches de traitement du langage naturel (NLP). Entraînés sur des ensembles de données textuelles volumineux, ces modèles montrent une habileté exceptionnelle dans diverses tâches, telles que la compréhension du langage naturel, la génération de texte, la traduction de langues et la génération de code. Dotés de milliards de paramètres ajustables, ils sont capables de saisir des motifs et des relations complexes dans les données textuelles. L'entrée d'un LLM est désignée sous le nom de «prompt», une série de tokens de longueur limitée.

L'amélioration des modèles existants pour des tâches particulières, appelée affinage *(fine-tuning)*, constitue une option plus abordable que l'entraînement complet des modèles. Il est difficile et coûteux de créer un LLM, ce qui demande beaucoup de temps et de ressources. En revanche, l'affinage de modèles déjà existants pour des tâches spécifiques

implique d'ajuster leurs paramètres en utilisant des données d'entraînement pertinentes, ce qui rend les modèles plus accessibles et permet d'obtenir des résultats optimisés. Après l'entraînement d'un modèle de langage par apprentissage supervisé, un modèle de récompense est développé et entraîné en utilisant des comparaisons basées sur les préférences humaines. Ce modèle de récompense permet d'optimiser le modèle de langage grâce à l'apprentissage par renforcement [39].

2.1.5 Transformers

La capacité des LLM à traiter d'énormes ensembles de données est au cœur de leur complexité ; un volume de données d'entrée plus important conduit à des sorties plus nuancées et précises. L'étiquetage méticuleux de ces données par des humains est essentiel pour améliorer les capacités des LLM. Cependant, récemment, ces modèles ont commencé à générer de manière autonome des ensembles de données étiquetées, simplifiant ainsi leur développement et réduisant le besoin de travail manuel. Les Transformers ont joué un rôle crucial dans cette évolution, considérés comme une innovation clé dans l'architecture des modèle NLP.

Contrairement aux modèles traditionnels comme les RNN (Réseaux de Neurones Récurrents) et les LSTM (Long Short-Term Memory), qui traitent les données de manière séquentielle, les mécanismes d'auto-attention [46] permettent au modèle de prendre en compte l'importance relative de chaque mot dans une phrase par rapport à tous les autres, indépendamment de leur position. Cela permet au modèle d'identifier les parties les plus importantes de la phrase pour comprendre le sens global, sans être limité par l'ordre des mots. Ainsi, le Transformer traite des séquences de texte entières en parallèle, offrant une efficacité et une rapidité accrues.

Cependant, les Transformers ne connaissent pas, par défaut, la position des mots dans la séquence. Pour remédier à cela, ils utilisent une information de position appelée intégration positionnelle (positional embedding). Cette approche permet au modèle de comprendre la position relative des mots dans la séquence, ce qui aide à différencier les tokens ayant le même contenu mais situés à des positions différentes, améliorant ainsi la capacité du modèle à comprendre et générer du texte de manière plus précise et contextuellement appropriée, tout en conservant le contexte des informations précédentes.

Ces avancées ont permis au Transformer de devenir un modèle incontournable pour de nombreuses tâches de compréhension du langage, surpassant largement la simple

traduction automatique. Il a également été à la base de nombreux autres modèles avancés dans le domaine du traitement du langage naturel (NLP). Les principaux composants d'un Transformer sont le module Encodeur-Décodeur, le module d'Auto-Attention et le module d'Embedding Positionnel, comme illustré à la figure 2.2.

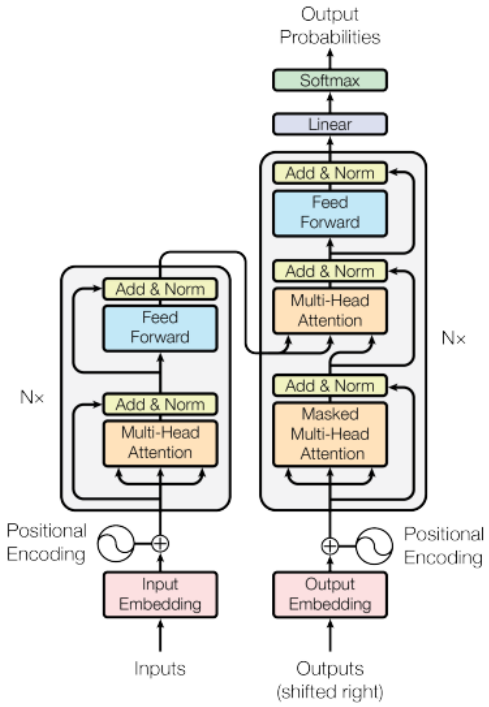

FIGURE 2.2 – Architecture des Transformers, tirée de l'article original [46]

2.1.6 Chatbot basé sur l'IA

En raison de l'intérêt grandissant pour les Chatbots, il est important de comprendre le fonctionnement de cette nouveauté technique qui est devenue un sujet brûlant dans le domaine de la technologie de l'information.

Un chatbot est principalement une application logicielle web qui permet de communiquer avec des individus en utilisant du langage textuel. Ces programmes exploitent les LLMs, qui sont des systèmes d'IA entraînés à comprendre et à générer du langage humain [15]. Grâce à cette technologie, les chatbots peuvent comprendre et répondre aux requêtes et aux entrées des utilisateurs de manière conversationnelle. Ces systèmes sont polyvalents et peuvent être configurés pour diverses tâches telles que le service à la clien-

tèle, où ils peuvent gérer des requêtes simples ou fréquentes, ce qui allège la charge de travail des équipes humaines. L'exemple de ChatGPT, un chatbot développé par OpenAI, illustre bien l'application avancée de cette technologie, capable de conversations très naturelles et engageantes.

2.1.7 ChatGPT

ChatGPT *(Chat Generative Pre-trained Transformer)* est un modèle de langage développé par OpenAI, lancé en novembre 2022. Il se distingue par ses compétences en conduite de conversations, et en création de réponses adaptées au contexte, représentant une avancée majeure dans le domaine de l'intelligence artificielle conversationnelle. L'interaction avec ChatGPT se fait via des prompts introduits soit sur son interface de chat en ligne, soit par un appel API.

OpenAI a proposé ChatGPT dans plusieurs configurations pour s'adapter à divers besoins et ressources computationnelles. Ces configurations diffèrent par le nombre de paramètres, la capacité de traitement des tokens, la rapidité et le coût, offrant ainsi aux utilisateurs la possibilité de choisir le modèle le mieux adapté à leurs exigences. Parmi les modèles de la famille GPT-3.5, GPT-3.5-turbo, qui pointe vers gpt-3.5-turbo-0125 au moment de nos tests, possède 175 milliards de paramètres avec une taille maximale de tokens de 16 385, et a achevé son apprentissage en septembre 2021 [3]. GPT-3.5-turbo a été sélectionné pour notre étude en raison de sa combinaison optimale de haute performance, de coût économique et de capacité large de traitement. De plus, il est particulièrement apprécié des développeurs pour sa capacité à comprendre et générer du code dans plusieurs langages de programmation. Le fait qu'il possède 175 milliards de paramètres signifie que le modèle utilise un vaste réseau de variables internes pour apprendre les relations entre les mots et les phrases dans les données d'entraînement. Cela nous permet d'évaluer sa capacité à générer des réponses précises et contextuelles en matière de sécurité du code.

OpenAI a utilisé la technologie des Transformers pour développer ses modèles GPT, conçus pour comprendre et générer du texte de manière avancée. Pour améliorer ChatGPT et le rendre plus aligné avec les intentions humaines, OpenAI a appliqué l'apprentissage par renforcement à partir des feedbacks des utilisateurs humains. Cette méthode utilise les retours humains comme fonction de récompense pour obtenir le comportement souhaité, éliminant ainsi le besoin de développer une fonction objective manuellement.

Le processus d'affinage de ChatGPT se compose de trois étapes. D'abord, le modèle est entrainé par apprentissage supervisé avec des données étiquetées, où un entraîneur IA montre le comportement souhaité. Ensuite, le modèle affiné génère plusieurs sorties à partir de prompts, qui sont classées par un autre entraîneur IA. Ces classements servent à entraîner un modèle de récompense. Enfin, GPT-3.5 est affiné en utilisant ce modèle de récompense, avec la récompense servant à mettre à jour le modèle à chaque itération.

2.1.8 Ingénierie de prompt

L'ingénierie de prompt [34] fait référence au processus de conception et d'optimisation des prompts ou instructions données à un modèle de langage afin d'obtenir la réponse ou le comportement souhaité. Cela implique de formuler des requêtes d'entrée de manière que le modèle de langage génère les résultats les plus pertinents et précis. Cette technique est cruciale pour exploiter pleinement le potentiel des LLMs comme GPT-3.5 et GPT-4 pour diverses applications, y compris la génération de codes, la traduction et la réponse aux questions liées à la sécurité logicielle.

2.1.9 Tokens dans le contexte des LLMs

Un token est une unité de texte, comme un mot ou une partie de mot, utilisée par les modèles de langage pour traiter les données textuelles. Pour GPT-3.5 Turbo [3], un token est une séquence de caractères qui peut représenter des caractères, des mots ou des morceaux de mots. Dans la structure GPT, une série de tokens sert d'entrée et/ou de sortie, permettant au modèle de gérer efficacement et de comprendre des entrées textuelles complexes pour proposer un résultat souhaité. La taille maximale de tokens dans une session diffère d'un LLM à un autre, ce qui influe sur les coûts et les temps d'exécution. Pour GPT-3.5 Turbo, la capacité maximale est de 16 385 tokens, incluant 4 096 tokens pour le résultat produit après la phase d'inférence ou génération de sortie, basée sur le prompt donné.

OpenAI propose une calculatrice en ligne, Figure 2.3, permettant de trouver le nombre de tokens et de caractères dans une séquence de mots, expliquant également le processus de tokenisation. La tokenisation consiste à diviser les textes en unités plus petites pour faciliter leur traitement par les modèles d'IA, réduisant les coûts de calcul et de mémoire [1]. Elle affecte également la qualité et la diversité des textes générés et peut être réalisée par diverses méthodes adaptées à la complexité des textes. En général,

un token équivaut à environ 4 caractères dans un texte en anglais courant. En termes de mots, cela représente approximativement $\frac{3}{4}$ d'un mot, ce qui signifie que 100 tokens correspondent à environ 75 mots [1].

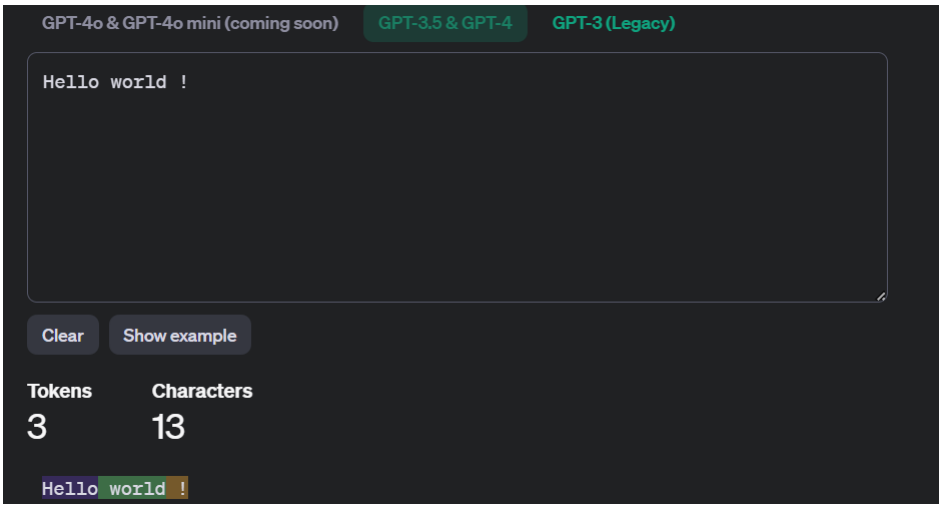

FIGURE 2.3 – Calculatrice de tokens pour les modèles GPTs -21/07/2024 [1]

2.2 La sécurité logicielle

La sécurité logicielle est un aspect spécifique de la sécurité informatique, distinct de la sécurité réseau, de la cryptographie ou de la sécurité des systèmes d'exploitation. Contrairement aux systèmes de détection d'intrusion ou aux antivirus qui préviennent les attaques de cybersécurité, la sécurité logicielle se concentre sur les failles présentes directement dans le code des applications, ce qui les rend vulnérables. Ces vulnérabilités doivent être corrigées rapidement pour éviter toute attaque malveillante.

2.2.1 Définition

Une définition plus précise de ce concept est donnée dans le livre de Khoury : *« La sécurité logicielle, ou sécurité des applications, se définit comme l'ensemble des pratiques, des vérifications et des mesures de précaution que les programmeurs devraient utiliser pour garantir que les systèmes qu'ils développent soient aussi sécurisés que possible.»* [32].

2.2.2 Vulnérabilités

Les vulnérabilités diffèrent des bogues car elles ne mènent pas toujours à une attaque. Si une vulnérabilité n'est pas exploitée, aucune attaque ne se produit. Selon la *National Vulnerability Database* (NVD) du *National Institute of Standards and Technology* (NIST) [17], une vulnérabilité est une«faille de sécurité, un bug ou une faiblesse dans le code logiciel qui peut être exploitée par un attaquant». Ces faiblesses peuvent découler de divers facteurs, tels que des erreurs de conception, des défauts de programmation, des configurations incorrectes ou des processus de gestion de la sécurité inadéquats. Lorsqu'une vulnérabilité est exploitée, elle peut permettre à des cybercriminels d'accéder à des informations sensibles, de perturber le fonctionnement du système, de voler des données ou d'exécuter du code malveillant, compromettant ainsi la confidentialité, l'intégrité et la disponibilité des ressources informatiques. Les vulnérabilités sont souvent classées selon la faiblesse sous-jacente qui les cause, même si elle n'est pas immédiatement visible. Cela permet d'identifier la catégorie d'erreur susceptible de mener à une faille de sécurité.

2.2.3 Vulnérabilités et Expositions Courantes

Les Vulnérabilités et Expositions Courantes (ou *Common Vulnerabilities and Exposures*) (CVE) sont une liste de vulnérabilités et d'expositions de sécurité de l'information divulguées publiquement. La liste CVE est maintenue par la MITRE Corporation [4], une organisation à but non lucratif qui exploite des centres de recherche et de développement financés par le gouvernement fédéral américain. Elle est alimentée par une communauté de chercheurs, de fournisseurs et de parties prenantes, qui signalent de nouvelles vulnérabilités. Ces contributions sont examinées et reçoivent des identifiants CVE uniques pour une référence cohérente et efficace. Chaque entrée dans la liste CVE contient un numéro d'identification, une brève description et des références aux rapports de vulnérabilité et aux avis connexes. L'objectif de CVE est de fournir un identifiant standardisé pour les vulnérabilités afin que les produits et services de sécurité puissent être plus efficacement coordonnés et automatisés, ce qui facilite la référence au même problème par différents systèmes et outils.

2.2.4 Énumération des Faiblesses Courantes

L'énumération des Faiblesses Courantes(ou *Common Weakness Enumeration)* (CWE) [20] est un standard utilisé pour catégoriser les faiblesses des logiciels. Chaque faiblesse est identifiée par un identifiant spécifique et comprend une description détaillée, les plateformes concernées, les conséquences potentielles, des exemples concrets, et des correspondances avec les CVEs. La dernière version de ce standard, CWE 4.15, inclut 939 faiblesses réparties en plusieurs catégories, ce qui permet de mieux identifier et gérer les vulnérabilités en sécurité logicielle.

La Figure 2.4 présente un exemple de vulnérabilité, permettant une compréhension rapide des aspects critiques de manière structurée et visuellement attrayante. Elle inclut l'identifiant CVE unique (e.g., CVE-2023-12345), l'identifiant CWE associé (e.g., CWE-119 pour un Buffer Overflow), la sévérité avec le score CVSS (e.g., High avec un score de 9.8), ainsi qu'une description succincte du problème et de son impact potentiel[5].

Champ	Détails
CVE-ID	CVE-2023-12345
CWE-ID	CWE-119 - Buffer Overflow
Sévérité	High (CVSS Score: 9.8)
Description	Dépassement de tampon dans xyz permettant l'exécution de code arbitraire.

FIGURE 2.4 – Détails de la Vulnérabilité CVE-2023-12345

2.2.5 Base de Données Nationale des Vulnérabilités

La Base de Données Nationale des Vulnérabilités (NVD)[5], maintenue par le NIST, est une ressource essentielle qui intègre tout le contenu CVE disponible publiquement [5]. Contrairement au CVE qui fournit des informations de base sur les vulnérabilités, la NVD offre des données supplémentaires telles que des scores de gravité, des évaluations d'impact et des analyses détaillées. Elle utilise le Système de Notation de Vulnérabilité Commun (CVSS) pour aider les organisations à prioriser leurs réponses en fonction de l'impact potentiel et de l'exploitabilité des vulnérabilités. La NVD propose également des flux de données intégrables dans des outils de sécurité pour automatiser la gestion et la mitigation des vulnérabilités, ainsi qu'une base de données recherchable par identifiant CVE, mots-clés ou logiciels spécifiques. Le site web de la NVD offre une interface

de recherche, des métriques détaillées, et des mises à jour régulières avec de nouvelles vulnérabilités et leurs scores CVSS, aidant ainsi les organisations à mieux comprendre et prioriser leurs efforts de sécurité. Ensemble, le CVE et le NVD sont des ressources cruciales pour les professionnels de la cybersécurité travaillant à protéger les systèmes contre les vulnérabilités connues.

2.3 Revue de la littérature

Nous effectuons une revue systématique de la littérature sur l'utilisation de ChatGPT pour le développement de logiciels sécurisés, en abordant divers aspects tels que la génération de code sécurisé, la détection et la correction de vulnérabilités dans les codes qui lui sont fournis, ainsi que d'autres tâches liées à la programmation sécurisée. Plus précisément, nous classifions les articles collectés selon l'application de sécurité associée à ChatGPT, comme la génération de code sécurisé ou la détection de vulnérabilités dans des fragments de code, en offrant un résumé concis de ces études. Ce résumé décrit les techniques d'évaluation utilisées, détaille leurs principales caractéristiques et présente les résultats clés, permettant d'identifier les conclusions communes et les résultats contradictoires.

Dans le but de raccourcir et d'éviter de s'attarder sur les résumés des articles, afin que le rapport ne soit pas trop long, nous nous contentons de décrire la démarche de notre revue systématique et de citer la classification de ces articles en fonction de la tâche de sécurité et de la méthode d'évaluation, ainsi que de présenter les conclusions tirées de chaque groupe d'études. En revanche, nous fournissons un résumé détaillé de certains articles particulièrement en relation avec notre démarche d'évaluation dans la section des travaux connexes. Notre protocole de revue implique une séquence structurée d'étapes :

1. Formuler les questions de recherche qui guideront l'exploration des principales sources numériques, facilitant la découverte et la compilation de la littérature la plus pertinente.
2. Planifier une stratégie de recherche minutieuse pour assurer une couverture complète.
3. Établir des critères clairs d'inclusion et d'exclusion pour l'évaluation systématique des études identifiées.

4. Sélectionner les études pour une revue détaillée.
5. Extraire et intégrer systématiquement les données afin de tirer des découvertes éclairantes et significatives du corpus de la littérature.

Ce cadre méthodologique assure un processus de revue exhaustif et systématique, en conformité avec les lignes directrices établies pour des revues de littérature rigoureuses, comme recommandé par Kitchenham et Charters[31] et Petersen et al. [40].

2.3.1 Questions de recherche

La question principale à laquelle cette étude vise à répondre est la suivante : Quel rôle ChatGPT peut-il jouer dans le processus de développement de code sécurisé ? Cette question générale peut être divisée en huit sous-questions :

QR1 : Quelles techniques sont employées pour détecter les vulnérabilités dans les fragments de code générés par ou fournis à ChatGPT ?
QR2 : Quelle version de ChatGPT est utilisée pour ces études ?
QR3 : Quel ensemble de données est utilisé dans l'expérience ?
QR4 : Pour quel langage de programmation ChatGPT produit-il le code le plus sécurisé ouou effectue-t-il l'analyse de sécurité la plus efficace ?
QR5 : Quelle version de ChatGPT est la plus performante dans les tâches liées à la sécurité ?
QR6 : Quelle est l'efficacité de ChatGPT pour produire du code sécurisé ou analyser la sécurité d'un code fourni, comparativement à d'autres LLMs ?
QR7 : Le code généré par ChatGPT est-il conforme aux meilleures pratiques de sécurité établies ?
QR8 : Les capacités de ChatGPT à rechercher des vulnérabilités sont-elles supérieures à celles des outils d'analyse statique ?

En répondant à ces questions, nous pourrons consolider un aperçu concis des recherches les plus récentes dans ce domaine et nous concentrer sur celles qui ont réalisé des évaluations expérimentales.

2.3.2 Catégorisation des études sélectionnées

Les études que nous avons examinées ont utilisé une variété de méthodes pour évaluer l'efficacité de ChatGPT dans l'exécution de tâches de sécurité liées aux programmes logiciels. Nous structurons donc ce recensement des écrits en conséquence. Ces études sont catégorisées comme suit

A. Evaluation de la sécurité du code produit par ChatGPT

A.1 Prompt textuel direct. La première catégorie comprend les études qui transmettent des instructions textuelles explicites via l'interface de chat pour inciter ChatGPT à générer du code dans divers langages de programmation, puis à évaluer la sécurité du code qu'il a produit. Un prompt textuel direct est une commande explicite fournie au LLM pour exécuter une tâche spécifique, telle que générer un programme qui répond à certains objectifs. En utilisant cette approche, l'accent est mis sur la tâche elle-même plutôt que sur la structuration du prompt d'une manière particulière ou sur l'orientation du comportement du modèle de langage à travers le contenu ou le format du prompt.

La principale conclusion de cette étape indique que ChatGPT 3.5 produit fréquemment du code contenant des vulnérabilités. Cependant, plusieurs nuances doivent être prises en compte. ChatGPT démontre généralement une capacité à identifier les vulnérabilités du code lorsqu'on l'interroge directement sur ce sujet. De plus, il semble y avoir un potentiel important pour exploiter ChatGPT afin d'améliorer la qualité du code grâce à une approche itérative consistant à présenter au chatbot des messages d'avertissement ou d'erreur générés par des outils d'analyse de code. Il est également à noter que ChatGPT présente des performances inférieures lorsqu'il génère du code dans certains langages, particulièrement C/C++.

A.2 Prompt assisté. La deuxième catégorie d'études regroupe les articles utilisant un prompt assisté, c'est-à-dire que le prompt est soigneusement conçue en suivant un protocole spécifique, puis transmise à ChatGPT soit directement, soit à l'aide d'un outil intermédiaire, tel qu'un algorithme personnalisé, un framework, ou une API d'appel de fonctions. Cette approche fournit une entrée plus adaptée qui guide la sortie de ChatGPT vers le résultat souhaité. Notamment, diverses études ont adopté cette méthode et évalué les aspects de sécurité du code résultant. Dans la plupart des études, ChatGPT était très sensible à la qualité du prompt. L'utilisation d'un prompt structuré visant à guider le chatbot vers des réponses plus cohérentes et organisées donne de meilleurs résultats

dans divers langages de programmation. Ce constat souligne la nécessité de concevoir des prompt méticuleuses pour exploiter pleinement les capacités de cette technologie. De plus, l'intégration de connaissances spécifiques au domaine dans les prompts améliore l'efficacité de ChatGPT pour produire les résultats souhaités. Grâce à des prompts bien conçues incluant des connaissances pertinentes, ChatGPT dépasse les outils avancés pour la génération de tests de sécurité. Cependant, ChatGPT affiche une incohérence dans l'évaluation des implications éthiques de l'exécution des tâches liées à la sécurité, car il peut refuser de générer du code pour une tâche qu'il juge non éthique dans un cas, tout en produisant du code pour la même tâche dans un autre. Cela souligne la nécessité d'un cadre plus robuste et cohérent pour aborder les considérations éthiques associées aux tâches confiées à ChatGPT.

A signaler également que GPT-4 a surpassé GPT-3.5-Turbo en termes de performance en sécurité lors de la génération d'extraits de code [43]. Cela souligne l'importance de considérer la version spécifique d'un modèle de langage lors de l'évaluation des aspects de sécurité du code généré par l'IA.

B. Fournir à ChatGPT des fragments de code vulnérables

Dans cette section, nous examinons les études dans lesquelles le prompt fourni à ChatGPT inclut un fragment de code ou des méthodes vulnérables, éventuellement accompagné d'autres informations. ChatGPT est ensuite incité à rechercher des vulnérabilités dans le fragment, à corriger les vulnérabilités présentes ou à réaliser d'autres tâches liées à la sécurité. Les méthodes d'évaluation spécifiques employées dans ces cas pour mesurer l'efficacité de ChatGPT sont détaillées dans les trois sous-sections suivantes.

B.1 Génération de code correct à partir de code vulnérable. Ahmad et al [21] introduisent la méthode FLAG, où pour chaque ligne de code source vulnérable donné en entrée, un prompt avec préfixe et suffixe est créé pour fournir le contexte nécessaire aux modèles comme code-davinci-002 et gpt-3.5-turbo afin de générer un nouveau code ligne par ligne. Les lignes générées sont ensuite comparées aux lignes originales, et les divergences sont analysées pour détecter des vulnérabilités de sécurité ou des défauts fonctionnels.

B.2 Détection, classification et réparation de vulnérabilités. La majorité des recherches dans ce volet fournissent à ChatGPT des fragments de code et très souvent des méthodes vulnérables issus de différentes sources de données. Ces recherches utilisent

divers prompts aux contenus différents pour demander à ChatGPT de réaliser plusieurs tâches de sécurité. Ces études testent généralement chaque tâche de sécurité séparément, plutôt que de réaliser un scan complet du code. L'évaluation de l'efficacité peut se faire soit par une inspection manuelle, soit en utilisant des métriques d'évaluation standard de l'apprentissage automatique ou des métriques personnalisées. Les performances de ChatGPT sont souvent comparées à celles d'autres modèles de langage ou d'outils d'analyse statique pour tirer des recommandations sur son utilisation.

B.3 Détection de vulnérabilités dans les contrats intelligents. Enfin, un petit nombre d'articles se concentre spécifiquement sur l'utilisation de ChatGPT pour des tâches liées à la sécurité des contrats intelligents. Selon IBM[20], « Les contrats intelligents sont simplement des programmes stockés sur une blockchain qui s'exécutent lorsque des conditions prédéterminées sont remplies. » Plusieurs aspects des contrats intelligents, tels que leur immutabilité et leur décentralisation, rendent les préoccupations de sécurité particulièrement pertinentes dans ce contexte.

Les principaux résultats issus de cette investigation soulignent que ChatGPT montre une efficacité variable dans l'exécution des tâches de sécurité sur le code qui lui est fourni, selon les langages de programmation. Il est particulièrement efficace pour identifier et résoudre les problèmes de sécurité dans le code Python, offrant potentiellement des avantages par rapport aux outils établis couramment utilisés pour les tests de sécurité dans le développement logiciel. Cependant, des limitations apparaissent lorsqu'il s'agit d'analyser des codes en C/C++. Malgré ces limitations, les résultats mettent en évidence les performances exceptionnelles de ChatGPT dans certains types de vulnérabilités, en particulier les dépassements de tampon. En Java, des résultats contradictoires émergent de deux études. Cheshkov et al. [25] observent que ChatGPT a du mal à distinguer le code vulnérable de son code corrigé, ce qui entraîne un taux élevé de faux positifs.

Par rapport aux outils d'analyse statique existants, les résultats indiquent que la plupart des études axées sur la détection et la correction des vulnérabilités ont trouvé que ChatGPT surpasse les ces outils sophistiqués. De plus, la capacité de GPT à suggérer des correctifs pratiques est remarquable.

Toutefois, des résultats contradictoires apparaissent dans l'évaluation de la capacité de ChatGPT à détecter les vulnérabilités au travers de conversations multi-tours [24, 47]. Les résultats indiquent des limitations potentielles dans l'efficacité de ChatGPT en tant

que modèle d'IA pour la détection des vulnérabilités, en tenant compte de l'impact du contexte, y compris le jeu de données choisi.

Les outils basés sur les LLM [38, 44], dans lesquels ChatGPT est combiné avec des outils statiques, présentent de bons résultats dans la détection et la correction des vulnérabilités des contrats intelligents, mais ils ont également des limitations qui doivent être abordées, telles que la limite de tokens et l'incohérence des résultats lors de l'analyse de contrats de grande taille. En outre, l'approche en deux étapes, comme démontrée par GPTLENS [30], offre des avantages par rapport aux méthodes traditionnelles et obtient des taux de réussite plus élevés tout en éliminant le besoin d'expertise spécialisée dans le domaine de la sécurité des contrats intelligents.

2.4 Travaux connexes

Après avoir fait une revue profonde de la littérature, nous pouvons situer notre étude parmi les recherches utilisant ChatGPT 3.5, particulièrement GPT-3.5-Turbo pour l'analyse de la sécurité du code. Ces recherches partagent un objectif commun : évaluer la capacité de ChatGPT à analyser un code vulnérable pour accomplir diverses tâches de sécurité. Dans certains cas, les performances de ChatGPT sont comparées soit à d'autres modèles LLM, soit à des outils d'analyse statique pour en tirer des recommandations ou fournir des pistes d'amélioration.

Les recherches existantes, similaires à notre travail, varient principalement par les contextes utilisés dans l'expérimentation, tels que le jeu de données, le langage de programmation choisi, les métriques d'évaluation, le contenu des prompts, les appels API, ainsi que les méthodes de détection, de classification et de réparation des vulnérabilités. Par conséquent, le recensement de ces travaux permet de déterminer si les études arrivent à des conclusions similaires concernant les performances de ChatGPT dans des contextes différents ou si elles sont contradictoires.

Nous introduisons cette section avec une étude menée par Cheshkov et al. [25] qui évaluent la faisabilité d'utiliser des LLMs notamment Davinci et GPT-3.5-turbo, pour détecter et classer les vulnérabilités dans le code source Java avec une interaction via l'API d'appel de fonction d'OpenAI pour les modèles GPT. Ils examinent les performances des modèles dans des classifications binaires (code vulnérable vs corrigé) et multi-classes (identification des types spécifiques de vulnérabilités CWE). Pour cela, ils collectent 428 fichiers Java de GitHub, contenant à la fois du code vulnérable et cor-

rigé. Les résultats montrent que, pour la classification binaire, les modèles GPT ont des scores de rappel et de F1 plus élevés mais un score AUC similaire à un modèle aléatoire, indiquant une difficulté à distinguer le code vulnérable du corrigé. En classification multi-étiquettes, les modèles atteignent 38% de précision, mais la suppression des exemples négatifs (les codes corrigées) réduit cette précision, soulignant leur difficulté à identifier correctement les vulnérabilités sans contraste. ChatGPT est plus précis pour classer le code bénin, probablement en raison de son entraînement sur du code correct ou de la taille limitée de l'échantillon. Dans l'étude de Bakhshandeh et al.[22], l'objectif est d'évaluer ChatGPT pour détecter les vulnérabilités dans de codes sources Python. Leur méthodologie consiste à fournir au modèle GPT-3.5 Turbo du code vulnérable provenant de deux ensembles de données différents : le jeu de données securityEval introduit dans [42], comprenant 130 fichiers de code vulnérable, et le jeu de données PyT utilisé dans[35], comprenant 26 fichiers de code vulnérable. Pour les 156 fichiers de code Python collectés, ils identifient manuellement les lignes de code correspondant aux CWE assignées dans les étiquettes de ces fichiers. ChatGPT est ensuite chargé de découvrir toute vulnérabilité présente dans le code soumis via l'API d'appel de fonction d'OpenAI. Les résultats sont comparés à ceux obtenus par trois outils d'analyse de sécurité statique couramment utilisés, à savoir Bandit [19], Semgrep [16] et SonarQube [18]. ChatGPT s'est comparé favorablement à ces outils, montrant des taux de faux positifs et de faux négatifs plus faibles. Les résultats indiquent que ChatGPT pourrait être un outil précieux pour détecter les vulnérabilités dans le code source Python. Purba et al.[41] investiguent la capacité des LLMs à classifier des fragments de code pour deux vulnérabilités spécifiques : le débordement de mémoire tampon (buffer overflow) et l'injection SQL. Ils cherchent à évaluer l'impact du fine-tuning des LLM sur des données étiquetées avec des vulnérabilités connues et à identifier les caractéristiques que les LLM peuvent ou ne peuvent pas détecter lors de la prédiction des vulnérabilités. Les auteurs comparent quatre LLM : GPT-3.5-Turbo, GPT-4, le modèle Davinci et CodeGen [37], avec des outils standards de découverte de vulnérabilités tels que Flawfinder [11], RATS [12], Checkmarx [7], ainsi qu'un réseau neuronal personnalisé, Vuldeepecker [33]. Ils ont tiré des exemples de vulnérabilités d'injection SQL en Python de la base de données CVEfixes [23] et des exemples de vulnérabilités de dépassement de mémoire tampon en C/C++ à partir de Code Gadgets[33]et CVEfixes [23]. L'étude utilise l'API d'OpenAI pour interroger ChatGPT et le fine-tuning pour les modèles Davinci et CodeGen. Les auteurs ont utilisé trois formulations différentes d'un prompt direct, demandant la

présence d'une vulnérabilité dans un fragment de code donné. Les résultats pour les trois variantes étaient similaires et généralement décevants. Les LLM permettant le fine-tuning pour la tâche demandée surpassent ceux qui ne le permettent pas, bien que les faux positifs restent élevés. Les LLM montrent de meilleures performances sur le jeu de données, Code Gadgets comparé à CVEfixes, un résultat que les auteurs attribuent à une étape de prétraitement de découpage de code effectuée sur le premier jeu de données. Le découpage de code met en évidence les parties.les plus pertinentes du code, facilitant la tâche du LLM. L'étude révèle une disparité significative de performance entre les deux types d'outils, en grande partie due à des taux plus élevés de faux positifs des LLM par rapport aux outils d'analyse statique traditionnels. Néanmoins, les LLM démontrent un potentiel dans l'identification de motifs complexes associés aux vulnérabilités logicielles, indiquant un avantage possible à combiner les LLM avec des méthodes d'analyse de programme supplémentaires pour améliorer la détection des vulnérabilités. Une étude similaire sur le code C/C++ a été menée par Fu et al. [27]. Les auteurs évaluent ChatGPT sur quatre tâches de détection de vulnérabilités : identification, localisation, identification des types, estimation de la gravité et suggestion de correctifs. Deux versions de ChatGPT (gpt-3.5-turbo et gpt-4) sont comparées à quatre modèles de référence : AIBugHunter, CodeBERT, GraphCodeBERT et VulExplainer. L'évaluation se base sur 190 000 fonctions C/C++ tirées à partir des ensembles de données Big-Vul [26] et CVEFixes. Ils utilisent le jeu de données Big-Vul pour évaluer plusieurs tâches de sécurité : la prédiction des vulnérabilités, leur classification et l'estimation de leur gravité. Pour la réparation automatique des vulnérabilités, ils emploient à la fois les jeux de données Big-Vul et CVEFixes. L'étude met en avant l'importance d'adapter les prompts pour chaque tâche. Pour la prédiction de vulnérabilités, un prompt est envoyé à ChatGPT pour lui demander de classer les fonctions vulnérables. Pour la classification des vulnérabilités, ChatGPT doit identifier le CWE-ID de la fonction vulnérable à partir d'une liste prédéfinie de CWE incluse dans le prompt. Pour l'estimation de la gravité, le prompt inclut des instructions sur la version CVSS attendue. Enfin, pour la réparation automatisée, trois échantillons des fonctions corrigées pour sont fournis pour guider ChatGPT dans la création de correctifs. Par la suite, ChatGPT est invité à créer des correctifs pour des fonctions vulnérables en utilisant les modèles fournis. Les résultats montrent que ChatGPT n'a pas particulièrement bien performé dans cette expérience, se révélant moins performant par rapport aux outils existants, parfois de manière significative. Par exemple, pour la prédiction de vulnérabilités, ChatGPT a obtenu un

F1-measure de 10 % et une précision dans le top-10 de 25 %. Les valeurs correspondantes pour AIBugHunter sont de 94 % et 99 % respectivement. De même, la précision du modèle en classification des vulnérabilités est de seulement 13 % à 20 %, loin des 62 % à 65 % obtenus par les modèles de référence. Des résultats similaires ont été observés pour la prédiction de la gravité et la réparation des vulnérabilités. Ces résultats rejoignent ceux d'autres études antérieures qui ont trouvé que ChatGPT présente des limitations lorsqu'il s'agit d'analyser des bases de code C/C++.

La table 2.1 présente une comparaison des études utilisant GPT-3.5 Turbo pour l'analyse de la sécurité du code, en tenant compte de divers contextes et tâches de sécurité évaluées. Chaque étude est décrite en fonction des données de test utilisées, qu'il s'agisse de codes complets ou de fonctions spécifiques, et de la nature des échantillons (vulnérables ou bénins). Les colonnes indiquent également si une API a été utilisée et les tâches évaluées, telles que la classification binaire, l'identification des types de CWE, les lignes vulnérables, et la correction.

TABLE 2.1 – Comparaison des études utilisant GPT-3.5 Turbo pour l'analyse de la sécurité du code, en fonction des contextes et des tâches de sécurité évaluées (I : Identification, S : Sévérité, T : Type CWE, L : Lignes Vulnérables, C : Correction, Codes Complets(CC), Fonctions (F), Vulnérables (V), Bénins (B), Vulnérables et Bénins(VB))

Papier	**Dataset**	**Taille**	**Fichiers contenant des :**	**Appel API**	**Tâche**
[25]	GitHub	428 fichiers Java	CC qui sont VB	Oui	I, T
[22]	[42] [35]	130 fichiers Python	CC qui sont V	Oui	I, T, L
[41]	[33] [23]	300 fragments en C/C++	F qui sont VB	Oui	T
[27]	[26] [23]	190 000 fonctions C/C++	F qui sont V	Oui	I, S, T, L, C
Notre approche	[23]	1248 codes en C/C++, Java, Python et PHP	CC qui sont VB	Oui	I, T, L, C

Notre approche se distingue des autres études mentionnées dans la table 2.1 par plusieurs aspects clés, notamment grâce à un contexte de test unique. Contrairement à Cheshkov et al. [25], qui se concentrent sur des fichiers Java tirés de GitHub et analysent uniquement les types CWE, et à Bakhshandeh et al. [22], qui évaluent des fichiers Python en se concentrant principalement sur l'identification des vulnérabilités et des types CWE, notre méthodologie couvre des codes complets en C/C++, Java, Python, et PHP, tirés de scénarios réels se trouvant dans un ensemble de données différent des leurs. De plus, nous analysons le code pour d'autres tâches de sécurité en plus de celles traitées dans leurs études.

Purba et al. [41] utilisent les jeux de données Code Gadgets et CVEFixes, se concentrant uniquement sur la classification de deux types de vulnérabilités, l'injection SQL et le dépassement de mémoire tampon, écrites en Python et C/C++ respectivement. Cependant, notre approche couvre plusieurs types de vulnérabilités en C/C++, Java, Python, et PHP. De plus, ils ne traitent pas d'autres tâches de sécurité telles que la localisation et la correction des vulnérabilités tout en préservant le contexte, comme nous le faisons. Notre méthode garantit une évaluation complète en tenant compte de tous ces aspects pour chaque code analysé.

Enfin, en comparaison avec Fu et al. [27], qui utilisent de nombreuses fonctions indépendantes en C/C++ et traitent diverses tâches de sécurité telles que l'identification, l'estimation de la sévérité, les types CWE et les corrections, notre étude se démarque par l'utilisation de codes complets, sans commentaires pour alléger le texte, en C/C++, Java, Python et PHP, provenant du jeu de données CVEFixes qu'ils utilisent également. Contrairement à leur méthode, nous avons focalisé nos efforts sur un seul jeu de données, évaluant toutes les tâches de sécurité sur un code particulier, plutôt que de tester certaines tâches avec un jeu de données et d'autres avec un autre. Ainsi, nous maintenons une cohérence en utilisant un seul ensemble de données pour toutes les évaluations.

De plus, nous avons développé une méthodologie d'analyse qui examine le code dans son intégralité tout en préservant le contexte. Nous simulons un dialogue continu avec le LLM tout en respectant la limite des tokens autorisés. Lors de l'analyse initiale, si des faux positifs (le modèle ne détecte pas une vulnérabilité qui existe réellement) ou des faux positifs (le modèle trouve une vulnérabilité du mauvais type CWE) sont détectés, nous effectuons une vérification supplémentaire pour confirmer l'existence réelle de vulnérabilités. Cette vérification est facilitée par la sauvegarde des réponses du LLM pendant l'interaction, permettant une analyse immédiate des résultats. Pour cette deuxième analyse, nous utilisons un prompt spécifique avec des indications supplémentaires et comparons les résultats obtenus avant et après la date d'entraînement du LLM.

Ainsi, notre approche, en maintenant le contexte et en réévaluant les fausses prédictions, offre une profondeur d'analyse qui n'est pas présente dans les autres études.

Chapitre 3

Méthodologie

Pour répondre à la problématique de cette étude, qui consiste à évaluer la sécurité et la fiabilité des résultats générés par ChatGPT dans le cadre de tâches de programmation sécurisée, ce chapitre présente notre étude expérimentale fondée sur une méthodologie rigoureuse. Ce processus, structuré en plusieurs étapes, comprend la collecte de données pertinentes issues de projets réels, la rédaction de prompts adaptés, la mise en place d'un dialogue interactif avec le modèle, ainsi que la collecte et l'analyse des réponses obtenues. La Figure 3.1 illustre le schéma de notre méthodologie, que nous détaillerons dans ce chapitre.

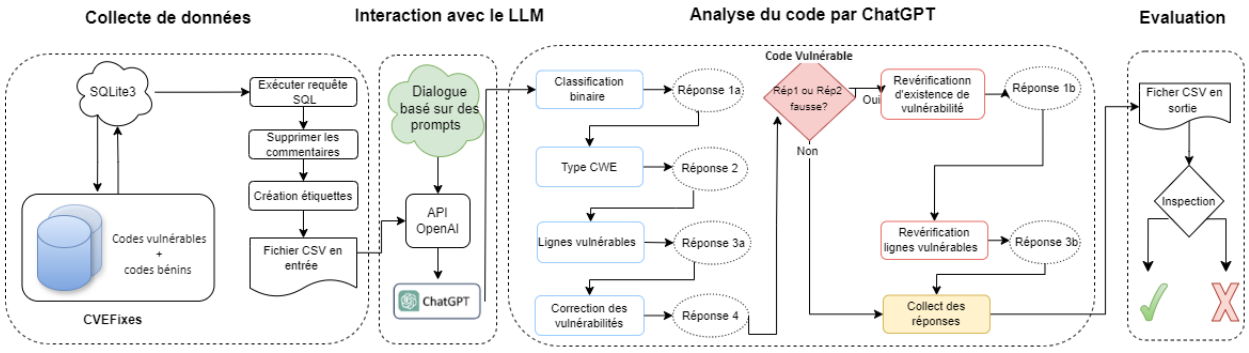

FIGURE 3.1 – Architecture de notre système

3.1 Collecte de données

Afin de mener notre expérience, il est nécessaire de collecter de réelles vulnérabilités pour évaluer si ChatGPT parvient à les détecter, les classer et proposer des correctifs. Pour atteindre cet objectif, nous avons opté pour la base de données CVEFixes [23].

3.1.1 Description de la base de données

CVEFixes est une base de données relationnelle en SQLite3, compilée automatiquement à partir de 5 365 enregistrements de vulnérabilités et expositions courantes (CVE) issus de la National Vulnerability Database (NVD) [5]. Il s'agit d'un système de données de gestion des vulnérabilités géré par le National Institute of Standards and Technology. Le NIST offre à l'usage public l'intégralité de la base de données NVD en utilisant différents flux de données, y compris des flux de vulnérabilité au format JSON qui sont mis à jour quotidiennement, organisés par année. Une fois qu'une vulnérabilité dans un projet open-source est résolue, l'enregistrement CVE sera actualisé avec un ou plusieurs pointeurs indiquant les dépôts de code source appropriés, ainsi que les hachages des commits des correctifs.

La Figure 3.2 représente la structure globale de la base de données CVEFixes, contenant les principales tables capturant des informations sur les CVEs, les CWE, les commits, les fichiers modifiés et les méthodes modifiées. Deux tables supplémentaires lient les correctifs et les types CWEs aux CVEs [23].

Le jeu de données contient le contenu des fichiers avant et après la modification, dans code_before et code_after respectivement, provenant de projets open-source. Il inclut également la différence des modifications au format fourni par Git, ainsi qu'une version analysée de cette information dans diff_parsed, contenant un dictionnaire des lignes ajoutées et supprimées. Plusieurs métadonnées sont collectées, telles que le nom du fichier, les anciens et nouveaux chemins, le type de modification, le nombre de lignes ajoutées ou supprimées dans ce fichier. Il comprend des fragments de codes couvrant plusieurs langages de programmation, notamment C/C++, Java, Python, R, Matlab et JavaScript, extraits de 5 495 commits de correction de vulnérabilités. L'édition initiale de CVEFixes a eu lieu en 2021, mais nous avons choisi la dernière version v1.0.7 mise à jour le 28 août 2022.

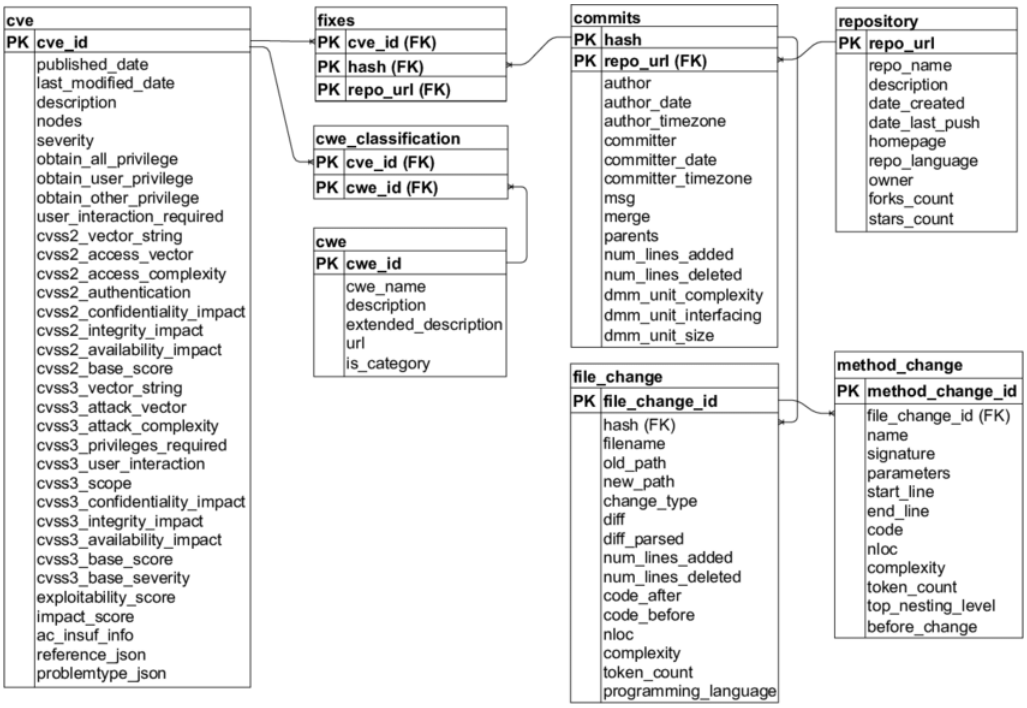

FIGURE 3.2 – Le schéma de la base de données CVEFixes extrait de l'article [23]

Nous avons choisi cette base de données pour les raisons suivantes :

1. Elle regroupe de véritables vulnérabilités tirées de projets réels à travers divers langages de programmation, offrant ainsi une référence pertinente pour notre analyse.
2. Elle contient plusieurs paires de codes sauvegardées dans des fichiers, comprenant une version vulnérable et sa version corrigée, ainsi que la CWE correspondante.
3. Elle rassemble une multitude de CVEs à travers plusieurs années, couvrant le premier CVE publié en 2002 jusqu'au dernier publié dans sa mise à jour d'août 2022. Cela nous permettra de tester ChatGPT à la fois sur des vulnérabilités qu'il a déjà rencontrées lors de son entraînement terminé en septembre 2021, ainsi que sur celles qu'il n'a pas encore vues, afin de capturer son comportement et ses performances.

3.1.2 Méthode d'extraction

Comme mentionné précédemment, CVEFixes est une base de données relationnelle. Pour l'interroger et extraire des données, il est nécessaire d'utiliser des requêtes SQL.

Pour procéder à cette extraction, nous avons d'abord installé la base de données CVEFixes en local. Nous avons utilisé DB Browser pour SQLite [8] afin de la visualiser et de pouvoir l'interroger via des requêtes SQL.

L'exploration de la base de données a permis d'identifier certaines contraintes à prendre en considération, qui constitueront nos conditions de sélection et permettront en partie un nettoyage de données. Ces dernières peuvent être résumées par les points suivants :

1. Les vulnérabilités dans la base de données CVEFixes varient en complexité, allant de corrections simples dans une seule méthode ou un seul bloc à des corrections plus complexes nécessitant plusieurs modifications dans un ou plusieurs fichiers. Pour évaluer les capacités de ChatGPT, nous avons ciblé les vulnérabilités comportant un seul fichier modifié. Cela permet principalement de mesurer s'il réussit à identifier, classer et corriger efficacement les vulnérabilités présentes dans un seul fichier, sans élargir l'étude à des vulnérabilités plus complexes impliquant plusieurs fichiers, surtout à cause des limitations en termes de tokens qui empêcheraient de fournir au LLM plusieurs fichiers nécessitant plus d'un code à analyser.

2. Filtrer les codes vulnérables ou corrigés dont la valeur est égale à 'None', c'est-à-dire éliminer les fichiers ne contenant que le code vulnérable ou que le code corrigé, car chaque fichier dans cette base de données contient à la fois le code vulnérable et sa version corrigée. Exemples : Le code vulnérable est None alors que le code corrigé existe pour la vulnérabilité CVE-2017-0576, et le code vulnérable existe mais le code corrigé est 'None' pour la vulnérabilité CVE-2015-1560.

3. Écarter les CWE avec les valeurs : NVD-CWE-Other et NVD-CWE-noinfo. Ces valeurs indiquent que la vulnérabilité n'a pas été classifiée de manière spécifique ou qu'il n'y a pas d'information disponible sur le type exact de vulnérabilité. En les écartant, on se concentre uniquement sur les vulnérabilités pour lesquelles une classification claire et détaillée est disponible.

4. Filtrer sur les langages de programmation visés, à savoir C/C++, Java, PHP et Python.

5. Filtrer sur le nombre de lignes supprimées pour corriger le code. La base de données indique la méthode de correction des vulnérabilités en précisant le nombre de lignes supprimées ou ajoutées. Les lignes supprimées représentent les lignes vulnérables et sont précédées d'un signe « - », tandis que les lignes ajoutées pour corriger le code sont précédées d'un signe « + ». Ces lignes sont traitées et analysées, puis stockées dans

Diff_parsed. Dans certains cas, une vulnérabilité peut exister dans le code sans qu'il y ait une ligne vulnérable explicite, par exemple lorsqu'il manque une déclaration de variable ou une ligne de code. Cela signifie qu'il n'y a pas réellement de ligne vulnérable supprimée. C'est pourquoi, dans la base de données, le nombre de lignes supprimées peut être égal à 0, indiquant que la correction de cette vulnérabilité se fait uniquement par l'ajout de nouvelles lignes de code. Il est donc nécessaire d'éliminer les fichiers où le nombre de lignes supprimées est égal à 0 afin d'avoir une réelle référence de comparaison des réponses du LLM en termes de localisation des lignes vulnérables et de proposition de correctifs. De cette manière, nous conservons uniquement les codes vulnérables dont les lignes vulnérables sont connues à l'avance.

Ainsi, pour construire notre ensemble de données, nous avons méticuleusement élaboré une requête SQL respectant les critères et contraintes cités ci-haut afin de satisfaire nos objectifs tout comme le montre la figure 3.3.

```sql
SELECT
    cv.cve_id,
    cv.published_date,
    f.filename,
    cc.cwe_id,
    f.code_before,
    f.code_after,
    f.diff,
    f.diff_parsed,
    f.programming_language,
    f.num_lines_added,
    f.num_lines_deleted

FROM
    file_change f
JOIN
    commits c ON f.hash = c.hash
JOIN
    fixes fx ON c.hash = fx.hash
JOIN
    cve cv ON fx.cve_id = cv.cve_id
JOIN
    cwe_classification cc ON cv.cve_id = cc.cve_id
WHERE
    cv.cve_id IN (
        SELECT cv.cve_id
        FROM file_change f
        JOIN commits c ON f.hash = c.hash
        JOIN fixes fx ON c.hash = fx.hash
        JOIN cve cv ON fx.cve_id = cv.cve_id
        JOIN cwe_classification cc ON cv.cve_id = cc.cve_id
        GROUP BY cv.cve_id
        HAVING COUNT(*) = 1
    )
    AND cc.cwe_id NOT IN ('NVD-CWE-Other', 'NVD-CWE-noinfo', 'CVE-2019-2080€')
    AND cv.cve_id NOT IN ('CVE-2017-0576', 'CVE-2015-1560')
    AND f.programming_language IN('C', 'C++','Java','Python','PHP')
    AND f.num_lines_deleted != 0
    AND LENGTH(f.code_before) <= 55000
    AND LENGTH(f.code_after) <= 55000;
```

FIGURE 3.3 – La requête SQL utilisée pour extraire les données à partir de la base de données CVEFixes

Nous avons ensuite sauvegardé le résultat de cette requête dans un fichier au format CSV, obtenant ainsi 1248 fichiers qui constitueront des données de test pour évaluer le

LLM. Chaque enregistrement dans ce fichier comprend toutes les informations importantes relatives aux vulnérabilités extraites, telles que l'identifiant de la vulnérabilité (CVE_ID fourni par la NVD), sa date de publication, son type CWE, le nom du fichier contenant le code vulnérable et son correctif, le code avant (vulnérable), le code après (corrigé), les lignes vulnérables, le langage de programmation, et d'autres attributs. La figure 3.4 représente un extrait de ce fichier.

Nous avons extrait des codes en C/C++, Java, PHP et Python de la base de données CVEFixes. Sur un total de 1248 fichiers, nous avons rassemblé 678 fichiers en C, 137 en C++, 61 en Python, 328 en PHP et 44 en Java, comprenant à la fois des versions vulnérables (code_before) et des versions sécurisées (code_after).

cve_id	filename	published_date	cwe_id	code_before	code_after
CVE-2002-2443	schpw.c	2013-05-29T14:29Z	CWE-20	#include "k5-int.h"…	#include "k5-int.h"…
CVE-2014-5353	ldap_pwd_policy.c	2014-12-16T23:59Z	CWE-476	#include "ldap_main.h"…	#include "ldap_main.h"…
CVE-2014-9422	kadm_rpc_svc.c	2015-02-19T11:59Z	CWE-284	#include <k5-platform.h>…	#include <k5-int.h>…
CVE-2015-2697	bld_princ.c	2015-11-09T03:59Z	CWE-125	#include "k5-int.h"…	#include "k5-int.h"…
CVE-2015-8629	kadm_rpc_xdr.c	2016-02-13T02:59Z	CWE-125	#include <gssrpc/rpc.h>…	#include <gssrpc/rpc.h>…

FIGURE 3.4 – Extrait du fichier d'entrée

3.1.3 Limitations du jeu de données

Il existe un attribut dans la base de données CVEFixes appelé token_count, utilisé pour compter le nombre de tokens dans le code. Au début, nous avions envisagé de l'exploiter pour filtrer et gérer la longueur excessive des fichiers de code dans les scénarios réels. En effet, lorsque le code est trop long, il peut entraîner, avec le prompt et les réponses, un dépassement du nombre maximal de tokens que LLM peut traiter en une seule fois. Cependant, nous avons découvert que cet attribut contient de nombreuses valeurs manquantes, avec un total de 180 valeurs marquées comme (None/NaN). De plus, les valeurs existantes dans ce champ ne reflètent pas exactement la taille du code en termes de tokens. En effet, token_count a été défini comme texte. Pour choisir les tailles souhaitées nous avons transformé le type texte en numérique dans la requête pour pouvoir les ordonner de façon décroissante : ORDER BY CAST (token_count AS UNSIGNED). Malheureusement, dans plusieurs situations, le compte n'était pas exact, car parfois le compte est inférieur au nombre de caractères diviser par 4. Par exemple, pour CVE-2021-3879 : token_count=1 alors que code_after contient 27 645 caractères.

Ainsi, ce champ ne s'est pas avéré utile pour la sélection des codes. Pour remédier à cela, nous avons modifié la requête en ajoutant deux conditions supplémentaires à la clause WHERE :

LENGTH(f.code_before) <= 55000 : Cette condition filtre les enregistrements où la longueur de la colonne code_before est inférieure ou égale à 55 000 caractères.

LENGTH(f.code_after) <= 55000 : Cette condition filtre les enregistrements où la longueur de la colonne code_after est inférieure ou égale à 55 000 caractères.

De la sorte, nous sélectionnons uniquement les enregistrements pour lesquels les codes vulnérables et leurs corrections ont une longueur maximale de 55 000 caractères chacun. Nous avons choisi 55 000 car la taille maximale du contexte que GPT-3.5-turbo peut gérer est de 16 385 tokens, équivalent à environ 65 540 caractères (16 385 * 4), en incluant le code, les prompts et les réponses. Donc, nous avons alloué 55 000 caractères pour les codes, laissant le reste pour les prompts et les réponses du LLM.

3.1.4 Prétraitement des données extraites

En plus d'un filtrage méticuleux lors de la création de notre ensemble de données, qui nous a permis de sélectionner des données complètes et d'éviter les valeurs manquantes, nous avons effectué un traitement supplémentaire pour les préparer à l'utilisation et les rendre plus adaptées à leur finalité.

Premièrement, pour alléger davantage la taille de notre fichier d'entrée et remédier aux limitations du nombre de tokens permis, nous avons créé un script Python qui supprime les commentaires des deux versions du code, vulnérable et corrigée, tout en préservant la structure initiale du fichier d'entrée.

Deuxièmement, pour tester le LLM sur les deux types de codes, vulnérable et non vulnérable, nous avons automatisé la création d'une nouvelle colonne nommée Target_code via un script Python dans le même fichier CSV issu de l'étape précédente. Cette colonne servira d'étiquette, contenant 70% de valeurs 1 pour désigner les codes vulnérables et 30% de valeurs 0 pour désigner les codes non vulnérables. L'attribution de ces valeurs est aléatoire et vise uniquement à distinguer le code vulnérable du code bénin. Elles n'introduisent en aucun cas un déséquilibre de données, car notre fichier d'entrée représente des données de test pour évaluer les capacités du LLM et non pas des données d'entraînement, où le rééchantillonnage est le plus couramment utilisé pour équilibrer les classes positives et négatives.

Nous pouvons en effet créer deux fichiers CSV différents, un pour les codes vulnérables et un autre pour les codes bénins, ou fournir le même fichier CSV deux fois, une fois pour les codes vulnérables et une autre fois pour les codes bénins. Toutefois, comme nous avons suffisamment de données et que le but principal est de vérifier si le LLM peut identifier, classer, localiser et corriger les vulnérabilités dans les codes connus comme vulnérables, les codes bénins ne sont fournis que pour vérifier s'il peut distinguer entre un code vulnérable et un code sain. Par conséquent, un petit échantillon de ce dernier, indiqué dans la colonne Target_code, est largement suffisant.

Remarque : l'ordre entre la suppression des commentaires et l'ajout de la nouvelle colonne Target_code n'est pas crucial.

Après ce prétraitement, nous avons obtenu 873 codes vulnérables et 375 codes bénins. La table 3.1 présente la répartition de ces codes parmi les différents langages de programmation sélectionnés.

TABLE 3.1 – Répartition des codes testés par langages de programmation

Langage de programmation	**C**	**C++**	**Java**	**Python**	**PHP**
Codes vulnérables	469	102	33	47	222
Codes bénins	209	35	11	14	106
Total	678	137	44	61	328

3.2 Interaction avec le LLM

Notre évaluation du modèle GPT-3.5-turbo de ChatGPT se fait via son API fournie par OpenAI, avec une documentation officielle [6] pour une utilisation efficace.

Le modèle GPT-3.5-Turbo peut accepter une série de messages comme entrée, au lieu d'un seul prompt de texte. Cela signifie qu'il peut traiter plusieurs messages consécutifs, ce qui est utile pour conserver le contexte des conversations et améliorer la cohérence des réponses. De plus, le modèle peut être configurer pour stocker des réponses antérieures ou faire des requêtes avec des instructions prédéfinies et du contexte, créant ainsi une sorte de dialogue continu. Cette fonctionnalité améliore significativement la qualité des réponses générées.

3.3 Désigne du prompt

L'échange avec un LLM se fait généralement à travers des prompts de commande. Il est donc essentiel de porter une attention particulière à leur format et à leur contenu.

3.3.1 La structure

ChatGPT est un LLM basé sur la complétion de chat pour générer des réponses à une série de messages dans une conversation. Via un appel API, il utilise des entrées spécifiques à configurer. Via un appel API, il utilise des entrées spécifiques à configurer, et le paramètre principal à considérer s'appelle «messages». Il est essentiel que les «messages» soient un tableau d'objets message, chaque objet ayant un rôle («system», «user», ou «assistant») et un contenu[6]. Les différents types de rôle dans les messages influencent le comportement du LLM. La table 3.2 décrit chaque rôle dans le message d'entrée.

TABLE 3.2 – Descriptions des différents rôles dans le message d'entrée .

Rôle	**Description**
Système	Permet de fournir un contexte au LLM afin de le préparer à recevoir des instructions spécifiques, mais il n'est pas obligatoire. Il se comportera de manière générique si ce message n'a pas été fourni.
Utilisateur	Contient les instructions ou commentaires que l'utilisateur adresse à l'assistant.
Assistant	Stocke les réponses précédentes de l'assistant ou peut être utilisé pour montrer des exemples de réponses souhaitées.

3.3.2 Le contenu

Notre objectif principal est d'évaluer les performances de ChatGPT dans l'analyse de la sécurité du code, en détectant, classant, localisant et proposant des correctifs pour les vulnérabilités identifiées. Nous visons à simuler une interaction instantanée et réactive entre le développeur et le LLM de manière automatique, tout en mesurant sa capacité à servir à la fois d'outil d'analyse statique et de correction automatique des vulnérabilités. À cet effet, nous concevons un dialogue qui envoie des messages, enregistre les réponses et l'historique des conversations pour préserver le contexte.

Pour atteindre cet objectif, nous avons conçu une série de prompts, chacun dédié à une tâche de sécurité mentionnée à la figure 3.5, avec la codification suivante : Prompt de précision de contexte (P0), Prompt de détection de vulnérabilités (P1), Prompt de classification des vulnérabilités (P2), Prompt de localisation des lignes vulnérables (P3), Prompt de proposition de correctifs (P4), et Prompt de revérification de vulnérabilité avec une indication spécifique de CWE (P5).

Ces prompts sont donnés au LLM selon une logique et un ordre particulier, alternant entre les rôles d'utilisateur et d'assistant dans les messages.

Une fois que nous avons fourni le contexte au LLM via le prompt P0, nous passons le prompt P1 et capturons la réponse. Ensuite, nous passons le prompt P2 et capturons la réponse, et ainsi de suite jusqu'à passer toutes les requêtes.

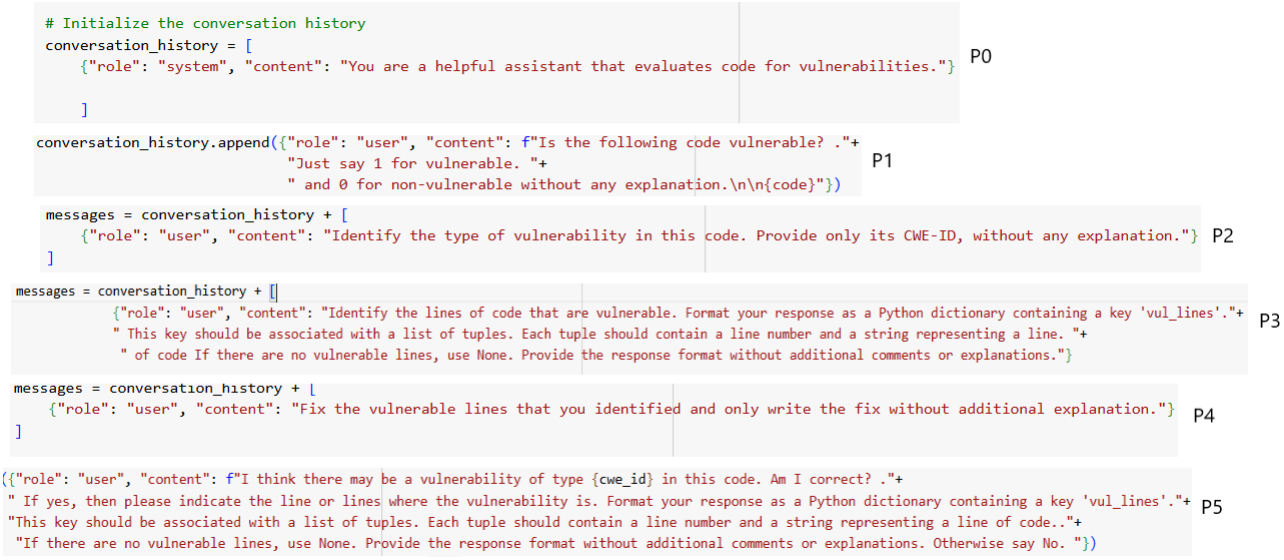

FIGURE 3.5 – Les différents prompts employés pour interagir avec ChatGPT

3.4 Expérimentation

Nous utilisons un environnement de développement Python pour implémenter un scénario simulant une conversation interactive avec GPT-3.5 Turbo via des appels API, nécessitant une connexion Internet stable et rapide pour mener à bien l'expérimentation. Chaque prompt, illustré à la Figure 3.5, et sa réponse sont gérés dans une méthode distincte afin de maintenir le contexte. Plus précisément, nous avons développé un script

qui automatise l'évaluation de GPT-3.5 Turbo pour analyser la sécurité du code, en récupérant ses réponses, qui constituent les résultats des analyses, afin de mesurer ses capacités et ses performances à identifier, localiser, classifier et corriger efficacement les vulnérabilités dans des fragments de code.

3.4.1 Scénario d'évaluation

Nous détaillons ci-dessous les différentes étapes constituant le scénario d'évaluation.

a. Initialisation

Nous commençons le processus par l'initialisation, où nous définissons la clé API OpenAI et créons un client, qui fait référence à un objet ou une instance pour interagir avec l'API d'OpenAI. Ce client permet d'effectuer des requêtes à l'API, d'envoyer des prompts et de recevoir des réponses. Ensuite, nous initialisons l'historique de la conversation avec une instruction système, représentée par le prompt P0 de la figure 3.5.

b. Configuration

La configuration inclut, d'une part, la mise en place de messages de journalisation (logging.info, logging.error) pour suivre l'état d'avancement du script. Cela permet d'identifier les points susceptibles de générer des erreurs, d'afficher des messages d'erreur pertinents et d'éviter que le script ne plante silencieusement. D'autre part, nous avons défini une fonction pour compter les tokens à chaque étape de l'exécution et vérifier leurs limites, car une conversation ne peut pas dépasser la capacité du modèle. Il est donc essentiel de raccourcir le contenu si nécessaire. Dans notre cas, nous avons veillé à fournir des codes complets sans les diviser en plusieurs fragments, tout en gérant la taille du contexte pour respecter les limites autorisées. Pour ce faire, nous avons prévu de réinitialiser le contexte à la fin du traitement de chaque ligne du fichier d'entrée contenant le code à analyser.

c. Traitement des fragments de code

Pour analyser les fragments de code, nous chargeons le fichier CSV contenant ces fragments et traitons une seule ligne à la fois. Pour chaque ligne, nous sélectionnons

le code à traiter en fonction de la valeur de Target_Code, en choisissant soit un code vulnérable, soit un code sain.

Si Target_Code = 1, cela signifie que le code que ChatGPT va analyser est vulnérable. Dans ce cas, nous évaluons le LLM sur toutes les tâches de sécurité, notamment l'identification, la classification, la localisation et la proposition de correction. Si Target_Code = 0, cela signifie que le code que ChatGPT va analyser est sain qui représente la version corrigée du code vulnérable. Dans ce cas, nous évaluons le LLM sur l'identification et la classification du type CWE s'il classe correctement le code comme non vulnérable. Sinon, en plus de l'identification et de la classification du type CWE, nous lui demandons de localiser les lignes vulnérables et de les corriger afin de mesurer s'il complexifie le code de façon non nécessaire en corrigeant un code non vulnérable. Nous décrivons ci-après le processus appliqué pour les deux cas :

Cas 1 : le code sélectionné est un code bénin

1. Envoyer le prompt P1 à l'API GPT-3.5 Turbo pour effectuer une classification binaire et vérifier si le code est vulnérable. L'objectif ici est de savoir si le LLM arrive à distinguer entre un code vulnérable et un code non vulnérable.

2. Obtenir la réponse du LLM (1 s'il identifie une vulnérabilité, 0 sinon, sachant que 0 est la réponse attendue).

3. Envoyer le prompt P2 pour obtenir l'ID CWE du type de vulnérabilité, quelle que soit la réponse du LLM, positive ou négative, afin de mesurer si le LLM parvient à identifier le CWE après la correction du code et s'il y a des éléments indicateurs qui le guideront dans sa réponse. Puis sauvegarder la réponse du LLM.

4. Si le code est identifié comme vulnérable (réponse égale à 1 pour un code non vulnérable) :

a. Envoyer le prompt P3 pour demander au LLM d'identifier les lignes de code vulnérables et de donner le nombre total des lignes puis sauvegarder la réponse du LLM.

b. Envoyer le prompt P4 pour demander au LLM de fournir une correction pour ces lignes. Puis sauvegarder la réponse du LLM.

5. Réinitialiser l'historique de la conversation ou le contexte et passer à la nouvelle ligne.

Cas 2 : le code sélectionné est un code vulnérable

1. Envoyer le prompt P1 à l'API GPT-3.5 Turbo pour effectuer une classification binaire et vérifier si le code est vulnérable. L'objectif ici est de savoir si le LLM arrive à classer correctement les codes vulnérables.

2. Obtenir la réponse du LLM (1 s'il identifie une vulnérabilité, 0 sinon, sachant que 1 est la réponse attendue).

3. Envoyer le prompt P2 pour obtenir l'ID CWE du type de vulnérabilité, quelle que soit la réponse du LLM, positive ou négative, afin de mesurer s'il parvient à trouver le CWE sans savoir identifier la vulnérabilité. Ensuite, sauvegarder la réponse du LLM.

4. Si le code est identifié comme vulnérable (réponse égale à 1) :

a. Envoyer le prompt P3 pour demander au LLM d'identifier les lignes de code vulnérables et de donner le nombre total des lignes. Puis sauvegarder la réponse du LLM.

b . Envoyer le prompt P4 pour demander au LLM de fournir une correction pour ces lignes. Sauvegarder la réponse du LLM.

5. Si le code n'est pas identifié comme vulnérable (réponse égale à 0) ou si l'ID CWE prédit ne correspond pas à la bonne réponse attendue, interroger à nouveau le LLM en lui envoyant le prompt P5 pour re-vérifier la vulnérabilité en fournissant un indice CWE et lui demander de confirmer s'il y a une vulnérabilité et de fournir les lignes vulnérables. Il est important de noter que l'indice CWE fourni dans le prompt représente le vrai type de vulnérabilité tel qu'attribué par la référence (*(ground truth)*, et il est récupéré automatiquement à chaque re-vérification. Sauvegarder la réponse du LLM.

6. Réinitialiser l'historique de la conversation ou le contexte et passer à la nouvelle ligne.

d. Enregistrement des résultats

Enfin, nous sauvegardons les résultats de l'analyse dans un nouveau fichier CSV qui regroupe toutes les données générées par ChatGPT, organisées dans des attributs créés au fur et à mesure pour capturer ses réponses. L'objectif principal de ce fichier est d'établir un lien entre les prédictions de ChatGPT et les références réelles à des fins de statistiques et d'analyse. C'est pourquoi nous y incluons les informations les plus importantes, telles que l'identification des vulnérabilités, le nom du fichier, et la véritable CWE et d'autres informations pertinentes. De cette manière, nous disposons d'un seul

fichier pour notre évaluation, contenant à la fois les données réelles et les prédictions du LLM. De plus, nous affichons les CVEs IDs ayant entraîné des problèmes durant le traitement afin de faciliter d'éventuelles analyses.

3.4.2 Métriques d'évaluation

Nous procédons dans cette section à l'évaluation des performances de ChatGPT. Cette évaluation consiste principalement à comparer les réponses de ChatGPT aux données réelles de référence stockées dans le fichier issu de l'étape précédente. Pour donner du sens à l'évaluation, il est crucial de choisir des métriques appropriées pour chaque tâche de sécurité effectuée par le LLM, comme expliqué dans la section 3.4.1.

a. Détection de vulnérabilités

Pour la détection des vulnérabilités, nous évaluons l'efficacité de ChatGPT à classer les codes comme étant vulnérables ou bénins. Il s'agit donc de réaliser une classification binaire. Dans les problèmes de classification, il est important de comprendre les différents types de résultats possibles pour évaluer correctement les performances d'un modèle. Dans notre cas, chaque ligne du fichier csv contient une étiquette dans la colonne Target_code, qui indique si le code fourni à ChatGPT est vulnérable ou bénin. En utilisant cette étiquette, nous pouvons définir des conditions positives et négatives pour analyser les performances du modèle. Dans ce contexte, Une condition positive indique le nombre de cas où le Target_code est 1, c'est-à-dire le nombre de codes vulnérables réels dans les données. Cependant, une condition négative indique le nombre de cas où le Target_code est 0, c'est-à-dire le nombre de codes bénins réels dans les données. En fonction de ces conditions, nous définissons quatre paramètres pour évaluer les performances de ChatGPT :

Vrai Positif (TP) : Le nombre de codes vulnérables correctement identifiés.

Vrai Négatif (TN) : Le nombre de codes bénins correctement identifiés.

Faux Positif (FP) : Le nombre de codes bénins incorrectement identifiés

Faux Négatif (FN) : Le nombre de codes vulnérables incorrectement identifiés.

Nous utilisons ces paramètres pour définir et calculer les métriques ci-dessous largement utilisées dans l'apprentissage machine :

Précision : La proportion des vrais positifs parmi tous les exemples étiquetés comme positifs par le modèle, calculée comme suit :

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.1}$$

Exactitude : La proportion de prédictions correctes par rapport au nombre total de prédictions effectuées, calculée comme suit :

$$\text{Exactitude} = \frac{\text{Nombre de prédictions correctes}}{\text{Total des prédictions}} \tag{3.2}$$

Rappel : La proportion des vrais positifs parmi tous les exemples réellement positifs dans les données, calculée comme suit :

$$\text{Rappel} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.3}$$

F-mesure : La moyenne harmonique de la précision et du rappel, calculée comme suit :

$$F_1 = 2 \times \frac{\text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}} \tag{3.4}$$

Taux de Faux Positifs (FPR) : FPR mesure la proportion de cas négatifs (bénins) qui sont incorrectement classés comme positifs (vulnérables) comme suit :

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{3.5}$$

b. Classification de la vulnérabilité selon le type CWE

L'évaluation des performances de ChatGPT pour l'identification des types de vulnérabilité (CWE) implique une classification multiple, car le LLM doit choisir un seul ID CWE parmi les différents types existants. Nous n'avons pas fourni de liste dans le prompt pour qu'il choisisse parmi celles-ci, mais nous lui avons laissé la liberté de fournir une réponse basée sur ce qu'il a appris durant sa période d'entraînement. De plus, chaque vulnérabilité dans le fichier de test a un type unique.

Nous souhaitons donc mesurer combien de fois il a bien identifié le type de CWE. Cela peut se faire simplement en calculant le nombre de prédictions correctes (où le type de vulnérabilité prédit par ChatGPT correspond au type réel) et de prédictions incorrectes

(où le type de vulnérabilité prédit ne correspond pas au type réel) en distinguant les codes vulnérables des codes non vulnérables.

Nous allons calculer l'exactitude pour prédire les CWE les plus récurrentes dans les données de test. L'exactitude est une métrique pertinente dans ce contexte, car elle permet de mesurer la proportion de prédictions correctes sur l'ensemble des cas testés, offrant ainsi une vue globale de la performance du modèle pour chaque classe de vulnérabilité (CWE).

c. Localisation des lignes vulnérables

L'analyse de code peut être réalisée à différents niveaux de granularité, notamment au niveau de la ligne, ce qui permet une évaluation précise des lignes spécifiques contribuant aux vulnérabilités. Cette approche offre un diagnostic détaillé, plus informatif que de simplement savoir qu'un code est vulnérable. Elle permet également aux développeurs d'apporter des corrections ciblées plutôt que de devoir réécrire ou modifier l'ensemble du code de manière significative. Pour comparer les lignes vulnérables prédites par ChatGPT à la référence, nous allons utiliser les métriques suivantes :

Intersection Over Union (IoU). Elle calcule l'intersection (nombre de lignes communes entre la référence et la réponse du LLM) divisée par l'union (Le nombre total de lignes uniques dans les deux ensembles combinés, la référence et le LLM.). IoU élevé (proche de 1) Indique une grande correspondance entre les lignes vulnérables prédites et les lignes vulnérables réelles. Un IoU de 1 signifie une correspondance parfaite. IoU faible (proche de 0) Indique une faible correspondance et suggère que les prédictions du modèle sont largement incorrectes.

Initial False Alarm (IFA). L'IFA est utilisée dans le contexte de la prédiction des défauts logiciels et de l'analyse des vulnérabilités. L'IFA mesure le nombre d'instances non défectueuses (faux positifs) rencontrées avant l'identification du premier véritable défaut (vrai positif). Cette métrique est particulièrement utile pour évaluer la performance des modèles conçus pour détecter les défauts ou les vulnérabilités tôt dans le processus de revue de code.

L'importance de l'IFA réside dans sa capacité à favoriser la détection précoce des véritables défauts avec un minimum de faux positifs, essentiel pour un tri efficace. Des valeurs élevées d'IFA peuvent nuire à l'expérience utilisateur, car les réviseurs doivent filtrer de nombreux faux positifs avant d'identifier un véritable défaut. L'IFA permet également d'évaluer et de comparer l'efficacité des modèles de prédiction de défauts.Une

faible valeur d'IFA (proche de 0) est souhaitable, car elle indique une détection rapide des vulnérabilités, tandis qu'une valeur élevée reflète une multiplication des fausses alertes, entraînant une perte de temps pour les analystes.

d. Correction automatique de vulnérabilités

Les correctifs suggérés par ChatGPT seront examinés manuellement pour garantir une analyse approfondie et précise de leur pertinence et efficacité. Pour l'évaluation, nous travaillerons sur des échantillons représentatifs de quelques-unes de ses réponses, ce qui permet d'éviter une comparaison automatique directe avec la référence, souvent moins pertinente dans le contexte de la génération de code. L'objectif n'est pas simplement de traiter le texte produit par le LLM en essayant de trouver des similitudes avec la version correcte, mais plutôt d'évaluer si le correctif proposé règle le problème.

En tenant compte de la variabilité des solutions, il est important de noter que la génération de code peut varier d'un développeur à l'autre. Cette diversité doit être considérée lors de l'évaluation des propositions de correctifs, car le LLM peut proposer un correctif qui peut être différent de celui existant dans la référence. De plus, cela nous permettra de vérifier si une complexité inhérente peut être introduite au code, ce qui peut significativement influencer la qualité et la pertinence des correctifs générés par le modèle.

Chapitre 4

Analyse des résultats

Dans ce chapitre, nous présentons les résultats détaillés de notre étude, basés sur les métriques précédemment définies et appliquées aux réponses obtenues. Nous analysons, d'une part, les performances de ChatGPT avant et après sa date d'entraînement et, d'autre part, ses capacités comparées à celles de deux autres LLMs, DeepSeek et Llama. Ce chapitre inclut également une section dédiée aux leçons apprises suite à l'utilisation de ces trois LLM en tant que réviseurs de la sécurité du code.

Commençons par analyser les résultats de ChatGPT avant et après sa phase d'entraînement, présentés dans les sections 4.1 à 4.5.

Nous avons constaté que ChatGPT n'a pas traité toutes les CVEs présentes dans le fichier d'entrée. Certains codes vulnérables ont posé problème lors du traitement, contrairement aux codes non vulnérables. Sur les 1248 enregistrements du fichier d'entrée, comprenant 873 codes vulnérables et 375 codes bénins, le LLM n'a pu traiter que 799 codes vulnérables, omettant ainsi un certain nombre de ces derniers. En revanche, il a traité tous les codes non vulnérables, 375 codes. Par conséquent, le fichier de sortie contient 1174 réponses correspondant au nombre de codes vulnérables et non vulnérables traités.

4.1 Détection de vulnérabilités

4.1.1 Classification binaire

Les résultats obtenus ci-dessous sont basée sur les deux classes, positives et négatives.

TABLE 4.1 – Classification binaire par langage de programmation avant et après septembre 2021.

	Jusqu'en septembre 2021			**Après septembre 2021**		
Langage	**Précision**	**Rappel**	**F1 Score**	**Précision**	**Rappel**	**F1 Score**
C	0.6599	0.9469	0.7778	0.7778	0.9608	0.8596
C++	0.71	0.9342	0.8068	0.9091	1	0.9524
PHP	0.6738	0.9474	0.7875	0.6078	0.9118	0.7294
Java	0.6552	0.95	0.7755	0.875	0.7778	0.8235
Python	0.8108	0.9091	0.8571	0.6875	0.9167	0.7857

D'après la table 4.1, ChatGPT démontre une bonne capacité à détecter les vulnérabilités et à classifier les codes vulnérables correctement par rapport aux codes non-vulnérables. Ses performances varient selon les données vues ou non vues durant sa phase d'entraînement, achevée en septembre 2021. Jusqu'à cette date, la précision varie entre 0.6552 et 0.8108, avec un rappel très élevé, indiquant une bonne couverture des vulnérabilités mais avec quelques faux positifs. Après septembre 2021, les performances s'améliorent pour certains langages comme C et C++, tandis que PHP et Python montrent une légère baisse de précision même avec un rappel assez élevé, suggérant une augmentation des faux positifs pour ces langages. Globalement, ChatGPT réussit bien à identifier les vulnérabilités, avec des scores F1 élevés, mais une attention particulière est nécessaire pour certains langages post-période d'entraînement. À cet effet, nous avons capturé séparément les taux de faux positifs dans la table 4.2 pour les deux périodes.

4.1.2 Classification des codes bénins

L'analyse des performances du modèle avant et après septembre 2021 (table 4.2) révèle des résultats contrastés selon les langages de programmation. Une amélioration modérée est observée pour le langage C, avec une diminution du taux de faux positifs (FPR) de 0.9485 à 0.9333, ainsi que pour Python, où le FPR passe de 0.875 à 0.8333. En revanche, une dégradation significative est constatée pour C++, où le FPR augmente de 0.8788 à 1, signalant une perte de précision. Le langage PHP affiche une performance

stable, avec un FPR quasi constant de 0.9531 à 0.9524. Enfin, pour Java, le FPR reste élevé à 1, indiquant un taux de faux positifs inchangé et préoccupant. Ces résultats suggèrent que les performances du modèle après septembre 2021 se sont améliorées pour certains langages tout en se détériorant pour d'autres.

TABLE 4.2 – Taux FPR par langages de programmation avant et après septembre 2021.

Langage	**FPR (Jusqu'en septembre 2021)**	**FPR (Après septembre 2021)**
C	0.9485	0.9333
C++	0.8788	1
PHP	0.9531	0.9524
Java	1	1
Python	0.875	0.8333

Ces résultats indiquent que la performance de ChatGPT est meilleure après septembre 2021 pour certains langages (C, Python), mais elle est pire pour d'autres (C++, Java), notamment avec une augmentation des faux positifs. Le modèle semble plus susceptible de signaler à tort du code bénin comme vulnérable, ce qui affecte sa précision. Un FPR de 1 pour certains langages indique une difficulté à distinguer correctement le code vulnérable du code non vulnérable. Cela suggère une nécessité d'améliorer les algorithmes ou les données d'entraînement postérieures à septembre 2021 pour une détection plus précise des vulnérabilités sans générer trop de faux positifs.

4.2 Classification des vulnérabilité selon le type CWE

4.2.1 Performances avant et après la date d'entraînement

La figure 4.1 illustre la répartition des prédictions correctes et incorrectes pour les types de CWE, distinguant les codes vulnérables des codes non vulnérables, à travers différents langages de programmation.

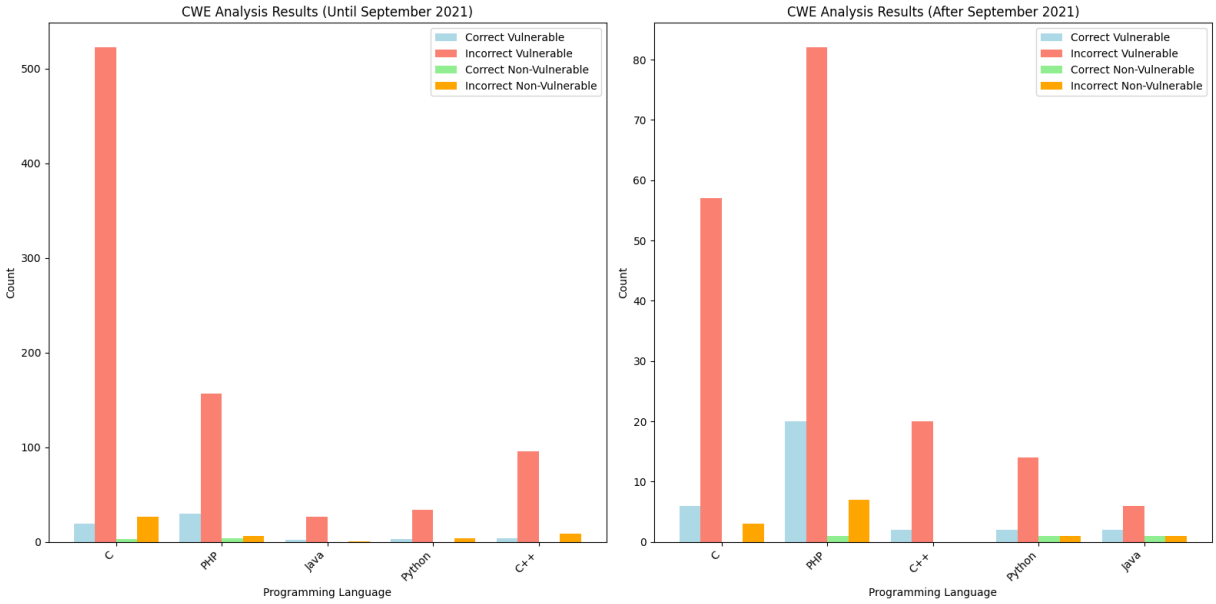

FIGURE 4.1 – Performances de ChatGPT pour la classification des vulnérabilités selon le type CWE avant et après la date d'entraînement.

Avant septembre 2021, le langage C se distingue par un grand nombre de prédictions incorrectes des types CWE, avec plus de 500 cas. Le langage PHP présente également un nombre élevé de prédictions incorrectes du type de CWE, avec plus de 100 cas incorrectes, reflétant des difficultés similaires à celles observées pour le langage C. En revanche, les prédictions correctes pour les codes vulnérables sont beaucoup moins fréquentes que les prédictions incorrectes, bien qu'il y ait quelques cas notables pour PHP et C.

Après septembre 2021, les prédictions incorrectes du type CWE demeurent prédominantes pour les langages C et PHP, bien que le nombre total de cas ait diminué, ce qui indique une amélioration partielle mais encore insuffisante. Cependant, une amélioration est observée dans les prédictions correctes pour les codes vulnérables, notamment pour PHP, contrairement aux codes écrits dans ce langage avant la phase d'entraînement de ChatGPT. Pour les langages Python et Java, il y'a moins de cas, mais des améliorations légères sont enregistrées en termes de prédictions correctes tant pour les codes vulnérables que non vulnérables.

En général, le LLM éprouve des difficultés significatives à identifier correctement le type de CWE pour les codes vulnérables, avec une tendance marquée aux prédictions

incorrectes, particulièrement pour les langages C et PHP. Malgré une amélioration après septembre 2021, les faux positifs restent élevés.

Les prédictions pour les codes non vulnérables demeurent également faibles, bien qu'une légère amélioration soit observée après septembre 2021 pour PHP et pour Java et Python. Pour ces deux derniers langages, le graphique montre une similarité dans la classification des CWE pour les codes non vulnérables, ce qui peut indiquer des faiblesses dans la capacité du modèle à gérer ces langages spécifiques en termes de prédiction correcte des CWE. Le modèle doit encore progresser dans l'analyse des codes propres.

4.2.2 Performances en fonction des CWE les plus récurrentes

Pour évaluer la performance de ChatGPT dans l'attribution du type CWE exact à une vulnérabilité, nous avons choisi d'identifier les types les plus récurrents dans les données de test afin de mieux comprendre les performances de ChatGPT sur les vulnérabilités les plus fréquentes.

Comme ChatGPT n'a traité que 1 174 CVEs parmi celles qui lui ont été fournies en entrée, nous avons donc calculé le nombre d'occurrence de chaque type CWE dans les CVEs traitées. Parmi les 1 174 CVEs analysées, 108 identifiants CWE uniques ont été répertoriés. Nous avons ensuite concentré notre analyse sur les 24 vulnérabilités les plus fréquentes, comme résumé dans la table 4.3.

Par exemple, les CWE-79 (*Cross-Site Scripting, ou XSS*), CWE-119 (*Improper Restriction of Operations within the Bounds of a Memory Buffer*), et CWE-125 (*Out-of-bounds Read*) apparaissent respectivement 158, 99, et 82 fois. À l'inverse, des CWE telles que CWE-287 (*Improper Authentication*), CWE-94 (*Improper Control of Generation of Code - Code Injection*), et CWE-400 (*Uncontrolled Resource Consumption*) sont moins fréquentes, avec seulement 11 à 12 occurrences chacune.

Après avoir identifié les différentes CWEs récurrentes, nous avons calculé l'exactitude (*accuracy*) du modèle. La Figure 4.2 illustre les fréquences ou occurrences de chaque type CWE ainsi que l'exactitude obtenue par le modèle pour chacune d'elles.

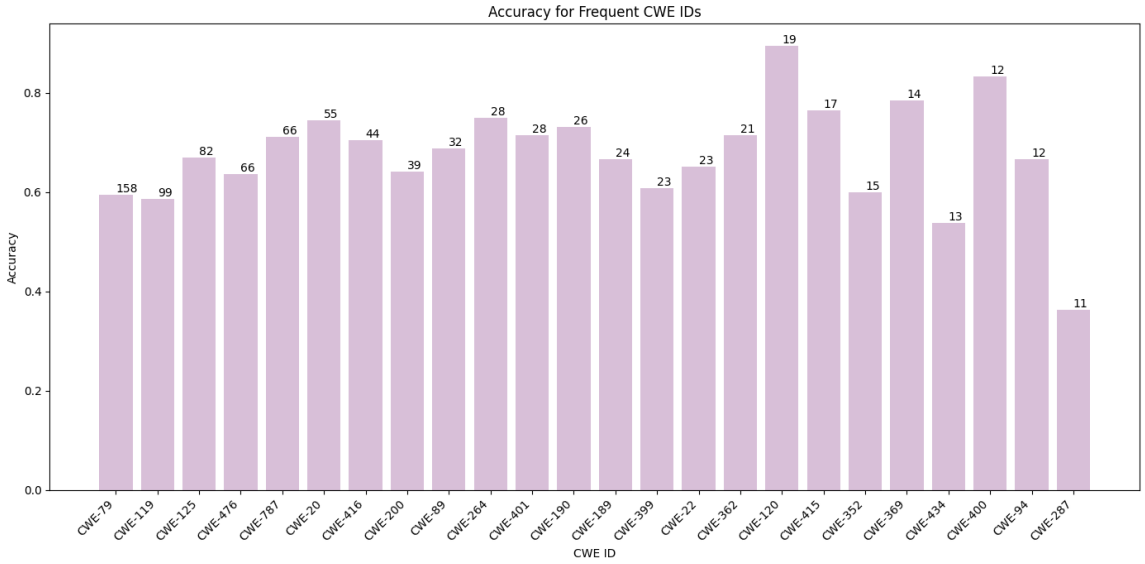

FIGURE 4.2 – L'exactitude de ChatGPT dans la classification des CWE récurrentes

Les résultats montrent que l'exactitude du modèle pour attribuer correctement les types de CWE aux vulnérabilités varie considérablement selon l'identifiant CWE. Par exemple, les CWE-120 et CWE-400 présentent des taux d'exactitude élevés (0,8947 et 0,8333 respectivement), indiquant que le modèle est relativement fiable pour ces types de vulnérabilités. En revanche, d'autres comme CWE-287 (0,3636) et CWE-434 (0,5385) montrent des performances inférieures. Les CWE-79 et CWE-119 figurent parmi les vulnérabilités les plus fréquentes, avec respectivement 158 et 99 occurrences. Cependant, les taux d'exactitude relativement bas de 0,5949 pour CWE-79 et 0,5859 pour CWE-119 suggèrent que le modèle a des difficultés à les identifier correctement malgré leur fréquence élevée. Ces variations montrent que l'efficacité du modèle dépend fortement du type de vulnérabilité analysé.

4.3 Localisation des lignes vulnérables

Nous avons observé que ChatGPT manque souvent de cohérence et de consistance dans ses réponses à des problèmes spécifiques et échoue fréquemment à respecter le format de réponse spécifié dans le prompt. Cette incohérence complique l'analyse automatique des résultats basée sur des métriques, et parfois, les données fournies ne sont pas précises, notamment lorsqu'il s'agit d'évaluer sa capacité à localiser les lignes vul-

TABLE 4.3 – Liste des CWE récurrentes analysées

ID	**Description**
CWE-119	Restriction incorrecte des opérations à l'intérieur des limites d'un tampon mémoire
CWE-120	Copie de tampon sans vérification de la taille de l'entrée ('Débordement classique de tampon')
CWE-125	Lecture hors limites
CWE-190	Dépassement d'entier ou Wraparound
CWE-20	Validation incorrecte des entrées
CWE-200	Exposition d'informations sensibles à un acteur non autorisé
CWE-22	Limitation incorrecte d'un nom de chemin à un répertoire restreint
CWE-287	Authentification incorrecte
CWE-352	Falsification de requête intersite (CSRF)
CWE-362	Exécution concurrente utilisant une ressource partagée avec synchronisation incorrecte ('Condition de course')
CWE-369	Division par zéro
CWE-400	Consommation non contrôlée des ressources
CWE-401	Libération manquante de mémoire après sa durée de vie effective
CWE-415	Double libération de mémoire
CWE-416	Utilisation après libération
CWE-434	Téléchargement non restreint de fichiers avec un type dangereux
CWE-476	Déréférencement de pointeur NULL
CWE-787	Écriture hors limites
CWE-79	Neutralisation incorrecte des entrées lors de la génération de pages Web
CWE-89	Neutralisation incorrecte des éléments spéciaux utilisés dans une commande SQL ('Injection SQL')
CWE-94	Contrôle incorrect de la génération de code ('Injection de code')
CWE-189	Erreurs numériques
CWE-264	Permissions, privilèges et contrôles d'accès
CWE-399	Erreurs de gestion des ressources

nérables. La figure 4.3 illustre les différentes manières dont ChatGPT répond lorsque nous lui demandons de lister les lignes vulnérables en format JSON—souvent sans respecter les normes. Parfois, il fournit des réponses avec des numéros de ligne, tandis qu'à d'autres moments, il répond avec le contenu des lignes.

FIGURE 4.3 – Exemples de formats variés de réponses de ChatGPT pour les lignes vulnérables

Pour mieux guider le LLM à répondre dans un format facilitant la comparaison avec les données de référence, nous avons utilisé l'ingénierie des prompts et créé un prompt plus détaillé,tel qu'illustré au prompt P3 de la figure 3.5. Nous avons demandé au modèle de fournir les réponses sous forme de dictionnaire Python afin de faciliter l'extraction des lignes prédites sous forme de chaîne de caractères, évitant ainsi de manipuler les numéros de lignes. En effet, se baser sur les numéros de lignes pourrait entraîner une invalidité des résultats, car la suppression des commentaires dans le code durant la phase de prétraitement des données peut affecter le nombre total de lignes et leur numérotation par rapport à ce qui existe dans les données de référence. De plus, la spécification d'un format précis pour les réponses permet d'éviter des réponses aléatoires et des explications inutiles.

Cette approche nous a permis d'appliquer les métriques IoU pour mesurer la proportion de lignes vulnérables correctement identifiées par rapport aux lignes réellement vulnérables, et IFA pour évaluer le nombre de lignes non vulnérables examinées avant de rencontrer la première ligne vulnérable. Un IFA élevé indique qu'un effort supplémentaire est nécessaire pour examiner les changements non vulnérables. Si la valeur de l'IoU est de zéro, cela indique que le modèle ne peut pas localiser correctement une seule ligne vulnérable. Si la valeur est de 1, cela signifie que le modèle peut prédire avec précision toutes les lignes vulnérables.

Pour chaque langage de programmation, les scores IoU et IFA ont été calculés pour le code vulnérable et non vulnérable, en tenant compte des données avant et après

septembre 2021 afin d'évaluer la capacité de GPT à localiser correctement les lignes vulnérables dans le code.

TABLE 4.4 – Mesure des indices IoU et IFA pour les codes vulnérables (V) et bénins (B) avant et après septembre 2021.

	Jusqu'en septembre 2021				**Après septembre 2021**			
	IoU (V)	**IFA (V)**	**IoU (B)**	**IFA (B)**	**IoU (V)**	**IFA (V)**	**IoU (B)**	**IFA (B)**
C	0.0007	8.381	0	8.8804	0	9.1224	0	9.4286
C++	0.005	6.0845	0	4.5517	0	8.45	0	1
PHP	0.0091	5.2143	0	5.541	0.0018	4.7419	0	6.875
Java	0	6.2105	0	2.7	0	5.8571	0	3
Python	0.0244	4.7	0	3	0	5.9091	0	1.2

Les résultats résumés dans la table 4.4 montrent que ChatGPT génère de nombreuses fausses alertes avant d'identifier une véritable vulnérabilité, comme l'indique la valeur élevée de l'IFA. En effet, une valeur d'IFA plus faible reflète une meilleure performance du modèle, car elle permet aux analystes de sécurité de détecter rapidement les premières lignes vulnérables sans être submergés par des faux positifs. Cela améliore à la fois l'efficacité et la rapidité des processus de révision de code et de gestion des vulnérabilités. En revanche, un IoU de 0 révèle que le modèle a réagi rapidement en signalant une vulnérabilité, mais a échoué, car la ligne signalée ne correspondait pas à une véritable vulnérabilité. C'est ce qui se produit pour les codes vulnérables, tant avant qu'après septembre 2021, où les valeurs de l'IoU sont nulles ou très proches de zéro pour tous les langages. La valeur la plus élevée, bien que faible, est de 0,0244 pour Python avant septembre 2021.

Il est aussi clair que, pour les codes bénins, les valeurs de l'IoU sont nulles dans tous les langages, que ce soit avant ou après la date d'entraînement de ChatGPT. Cependant, nous constatons la présence de valeurs d'IFA dans ces cas, avec des taux relativement faibles pour Java et Python, avant et après septembre 2021, ainsi que pour C++ après septembre. Cela ne signifie pas pour autant que le modèle est performant pour ces langages, car les valeurs nulles de l'IoU confirment que les lignes prédites ne sont en réalité que des fausses alertes.

Ce résultat met en lumière une anomalie intéressante dans le comportement de ChatGPT : le modèle a tendance à proposer des lignes comme vulnérables pour des codes déjà corrigés, ces lignes ne correspondant pas nécessairement à celles identifiées comme vulnérables dans la base de données avant correction. Ce comportement aléatoire

suggère à tort des lignes bénignes comme étant vulnérables, ce qui augmente le nombre de fausses alertes. Par conséquent, les développeurs peuvent être incités à effectuer des corrections inutiles, rendant le code plus complexe et entraînant une perte de temps et d'efforts supplémentaires.

Ainsi, la capacité du modèle à localiser précisément les lignes vulnérables reste extrêmement limitée, que ce soit avant ou après septembre 2021. Cela demeure un défi majeur pour ChatGPT.

4.4 Correction automatique de vulnérabilités

Nous effectuons une évaluation manuelle des correctifs suggérés par ChatGPT. Cependant, en raison du grand volume de données, il n'est pas possible de les évaluer toutes de cette manière, car cela serait très fastidieux. C'est pourquoi nous avons sélectionné certaines de ses réponses de manière ciblée, qu'il s'agisse de codes vulnérables ou bénins.

4.4.1 Correctif pour un code vulnérable

Nous souhaitons que la vérification de ces correctifs proposés pour de vrais code vulnérables serve non seulement à évaluer sa capacité globale à corriger des vulnérabilités, mais aussi à déterminer sa compétence à effectuer une analyse complète et efficace du code, fournissant ainsi un bilan positif ou négatif. Cela permettra de répondre à notre objectif d'analyse intégrale du code.

Pour vérifier l'existence d'un scénario correct ou d'un bilan complet, il est nécessaire d'extraire, à partir du fichier de sortie de chatGPT, un nouveau fichier contenant les réponses où il identifie correctement les codes vulnérables et attribue le type CWE de manière appropriée. Il faut ensuite évaluer les lignes vulnérables correctement ou partiellement identifiées, en considérant la correction des vulnérabilités comme le facteur déterminant. Par conséquent, le choix des codes à corriger doit correspondre à cette situation : si les corrections sont justes et sans complexité, le cas peut être classé comme un bilan positif.

À l'inverse, un bilan négatif reflète uniquement sur les cas où ChatGPT classe correctement un code bénin, indépendamment des autres réponses. Parmi les 375 codes bénins présents dans le fichier de réponses de ChatGPT, 22 ont été correctement classés, dont 4 avec des CWE exactes. ChatGPT a donc produit 22 bilans négatifs.

Concernant la vérification de l'existence de bilans positifs, un nouveau fichier a été créé contenant 74 réponses où ChatGPT identifie correctement les codes vulnérables et attribue le type CWE correctement. Ce nombre a été encore filtré pour déterminer quels correctifs suggérés par chatGPT méritent d'être évalués. Pour cela, nous avons d'abord analysé dans quelle mesure les lignes vulnérables proposées par le LLM pour ces cas étaient correctes. Les métriques IFA et IoU ont été appliquées pour évaluer le comportement du chatbot. La table 4.5 résume les résultats obtenus.

TABLE 4.5 – Mesure des indices IoU et IFA pour les codes vulnérables correctement dentifiés ainsi que leurs types CWE.

Vulnérable	**Jusqu'en septembre 2021**		**Après septembre 2021**	
	IoU	**IFA**	**IoU**	**IFA**
C	0	6.1875	0	4
C++	0	4.5	0	5
PHP	0	3.7826	0.0058	3.7368
Java	0	7	0	4.5
Python	0	1	0	3

D'après ces réslutats, tous les langages jusqu'en septembre 2021 présentent un comportement similaire, avec des valeurs élevées d'IFA, sauf pour Python, et des valeurs d'IoU égales à 0. Cela indique que les vulnérabilités signalées par le modèle ne sont pas correctement identifiées et ne constituent que de fausses alertes. Un IoU de 0 révèle une incapacité totale à localiser les lignes vulnérables, un problème commun à tous les langages durant cette période.

Après septembre 2021, l'IoU pour PHP s'améliore légèrement, passant à 0,0058, ce qui montre que ChatGPT a commencé à identifier correctement une très petite proportion des lignes vulnérables. Bien que modeste, cette progression démontre que le modèle a fait quelques avancées pour PHP, contrairement aux autres langages où l'IoU reste à 0. De plus, l'IFA pour PHP diminue à 3.7368, indiquant une très légère réduction du nombre de fausses alertes après cette période, ce qui reflète une amélioration, bien que limitée, dans la précision de la localisation des lignes vulnérables.

Ces résultats sont particulièrement pertinents pour cibler les correctifs à évaluer. Il est clair que l'analyse des correctifs se concentrera principalement sur les codes PHP écrits après la fin de la période d'entraînement de ChatGPT. Par conséquent, nous avons identifié 19 codes PHP à examiner.

L'examen a priori de ces codes révèle que, dans aucun cas, GPT ne parvient à fournir un correctif exacte de la vulnérabilité. Par conséquent, aucun bilan positif n'est enregistré. Une investigation plus approfondie sera nécessaire pour la prochaine étape, afin de mieux comprendre son comportement, évaluer dans quelle mesure il introduit des complexités supplémentaires, et inclure d'autres exemples pour chaque langage.

ChatGPT a tendance à corriger les fausses alertes en introduisant des structures et des conditions imbriquées. Parfois, il se contente de réécrire les lignes signalées comme vulnérables, sans apporter de vraie correction qui permet de résoudre la vulnérabilité.

Dans l'ensemble, des anomalies sont également constatées dans ses réponses, notamment lorsque le ChatGPT répond par "None" à un prompt demandant de corriger une vulnérabilité, malgré l'identification correcte du code comme vulnérable et la proposition de lignes candidates comme étant vulnérables. Nous avons enregistré 7 cas en C, 7 en PHP et 1 en Python où une réponse "None" a été donnée pour des codes identifiés comme vulnérables.

4.4.2 Correctif pour un code bénin

Pour les correctifs proposés à des codes bénins, nous avons constaté que GPT effectue souvent des corrections syntaxiques inutiles en ajoutant des «) », des «} » ou des opérateurs supplémentaires. Comme pour le point précédent, GPT ne propose pas de véritables corrections pour les lignes qu'il a identifiées. En revanche, il répond parfois par 'None'. Parmi ces cas, nous avons enregistré 9 codes en C, 5 en PHP, et un seul en C++, ce qui révèle une certaine incohérence dans ses réponses pour ce type de codes, confirmée plus haut par des résultats médiocres dans la classification binaire.

4.5 Re-vérification de l'existence de vulnérabilités

Parmi les 1174 réponses fournies par ChatGPT, nous avons identifié 725 cas où une re-vérification a été effectuée pour confirmer l'existence d'une vulnérabilité et localiser à nouveau les lignes vulnérables via le prompt P5, qui contient un indice sur le véritable type CWE présent dans le code. Cette re-vérification a été réalisée dans les cas où un code vulnérable avait été mal classé ou une CWE incorrectement attribuée à un code confirmé comme vulnérable.

Sur ces 725 cas, 78 (soit 10,75%) ont confirmé la présence de vulnérabilités avec une réponse positive de ChatGPT ("Yes"). En revanche, les 647 autres cas ont reçu une réponse négative ("No" ou "No."), Ce qui signifie que dans 89,24% des cas, la re-vérification n'a pas permis de réduire la prédiction aléatoire, c'est-à-dire des résultats générés sans analyse, déterminés uniquement par le hasard, ce qui reflète, dans certaines situations, l'inconsistance des réponses des LLMs face à des questions identiques.

Ces réponses nécessitent encore une analyse pour vérifier les nouvelles lignes vulnérables prédites. Pour cela, nous avons calculé les valeurs d'IFA et de l'IoU.

TABLE 4.6 – Performances de ChatGPT dans la detection de lignes vulnerables au second tour de vérification

Modèle	**avg_IoU**	**avg_IFA**
ChatGPT	0	1.5897

Les résultats de la table 4.6 montrent que ChatGPT n'a identifié aucune ligne vulnérable correcte lors de la phase de rectification en second tour. La valeur nulle de l'IoU confirme cette incapacité, tandis que la faible valeur de l'IFA indique que les rares lignes proposées sont des fausses alertes. Ainsi, le second tour de vérification n'a pas permis de réduire les erreurs dans les réponses du modèle.

4.6 Performances de ChatGPT comparées à celles d'autres LLMs

Dans les sections précédentes, nous avons présenté un aperçu des performances de ChatGPT dans diverses tâches de sécurité, en analysant sa capacité à traiter des vulnérabilités présentes dans les données sur lesquelles il a été entraîné jusqu'en septembre 2021, ainsi que de nouvelles vulnérabilités apparues après cette date et qu'il n'a pas vues.

Dans cette section, nous cherchons à comparer ChatGPT à d'autres modèles existants afin de mesurer ses performances face à d'autres LLMs. L'objectif est d'identifier ses points forts et ses faiblesses.

Pour ce faire, nous avons appliqué le même scénario d'évaluation utilisé pour ChatGPT, décrit dans le chapitre précédent 3 et illustré à la figure 3.1, à DeepSeek et Llama. Ces modèles ont reçu le même fichier de données en entrée que ChatGPT, contenant 1 248 CVEs, déjà adapté à la taille de contexte propre à chacun d'entre eux. Cela est particulièrement important, car ChatGPT possède la taille de contexte la plus petite des trois, et les données ont été choisies en conséquence pour s'adapter à cette limitation. Nous avons également utilisé les appels API spécifiques à chaque modèle et sauvegardé leurs réponses dans des fichiers distincts. Ci-dessous, la table 4.7 détaile chaque LLM et les résultats des CVEs qu'il a traitées :

TABLE 4.7 – Les modèles de langage étudiés

Modèle	**Version**	**Éditeur**	**Date**	**Taille**	**CVEs traitées**
ChatGPT	GPT-3.5 Turbo [3]	OpenAI	Nov 2022	175 milliards	1174
Llama	Llama 3.1 405B [13]	Meta	Juil 2024	405 milliards	573
DeepSeek	DeepSeek V2.5 [9]	DeepSeek	Sep 2024	236 milliards	1248

D'après la table 4.7, les CVEs traitées ne sont pas les mêmes pour chaque LLM. Malgré les ajustements effectués lors de la collecte des données, qui consistaient à choisir des données adaptées à la taille maximale de tokens permise par les trois modèles et à supprimer les commentaires, les CVEs traitées varient d'un modèle à l'autre.

Pour garantir une comparaison équitable sur les différentes tâches de sécurité visées, il est essentiel d'évaluer les modèles sur une base commune. À cette fin, nous avons extrait les CVEs communes traitées par les trois LLMs. Au final, nous avons retenu 543 CVEs, réparties en 364 codes vulnérables et 179 codes bénins.

Dans ce qui suit, nous détaillons les résultats de la comparaison entre ChatGPT, DeepSeek et Llama sur notre scénario d'évaluation.

4.6.1 Classification binaire

A. Performances globales. D'après la table 4.8, les résultats monterent que les trois LLM sont sur un pied d'égalité en termes de génération de faux positifs, car leur précision est faible et presque identique, autour de 67%. En comparant ChatGPT à DeepSeek et Llama, Llama excelle en rappel et en score F1 global, ce qui en fait le meilleur choix pour des tâches nécessitant un haut rappel et une performance équilibrée. ChatGPT offre un bon compromis avec un équilibre entre précision et rappel. DeepSeek sacrifie le rappel, ayant le plus faible des trois LLMs à 84%. Bien qu'il soit bon, il peut manquer certaines instances pertinentes.

TABLE 4.8 – Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans la classification binaire

Modèle	**Précision**	**Rappel**	**F1**
DeepSeek	0.674	0.8462	0.7503
Llama	0.6693	0.9451	0.7836
ChatGPT	0.6694	0.9066	0.7701

B. Performances par langage de programmation

Les résultats de la table 4.9 indiquent que Llama tend à surpasser ChatGPT et DeepSeek en termes de rappel pour la plupart des langages, avec des taux élevés, notamment en Java, C et C++ avec des valeurs respectives de 0.96, 0.94 et 0.94. Cela signifie qu'il identifie un plus grand nombre de vulnérabilités potentielles. Cependant, cette meilleure couverture n'entraîne pas toujours une augmentation significative du score F1, suggérant que Llama peut également produire plus de faux positifs.

ChatGPT obtient des scores compétitifs, mais reste inférieur à Llama en termes de rappel pour la plupart des langages.

De manière surprenante, en PHP, les deux modèles sont sur un pied d'égalité avec un rappel de 0.93 pour chacun. DeepSeek suit avec un rappel de 0.90, ce qui suggère que les trois LLMs excellent dans l'identification des vulnérabilités en PHP, un résultat rarement observé dans des études similaires. DeepSeek, quant à lui, présente des performances

TABLE 4.9 – Comparaison des performances de ChatGPT avec celles de Llama et de DeepSeek dans la classification binaire par langage de programmation

Langage	LLM	Précision	Rappel	F1	Total prédictions
C	DeepSeek	0.6164	0.7563	0.6792	190
	Llama	0.6243	0.9496	0.7533	
	ChatGPT	0.6213	0.8824	0.7292	
Python	DeepSeek	0.8056	0.8788	0.8406	41
	Llama	0.8	0.9697	0.8767	
	ChatGPT	0.8333	0.9091	0.8696	
PHP	DeepSeek	0.6751	0.9048	0.7733	221
	Llama	0.6618	0.932	0.774	
	ChatGPT	0.6618	0.932	0.774	
C++	DeepSeek	0.7111	0.8421	0.7711	54
	Llama	0.72	0.9474	0.8182	
	ChatGPT	0.7234	0.8947	0.8	
Java	DeepSeek	0.7273	0.8889	0.8	37
	Llama	0.7222	0.963	0.8254	
	ChatGPT	0.7059	0.8889	0.7869	

plus variables selon les langages, mais reste en général derrière les deux autres modèles. Cependant, en Java, il atteint le même rappel que ChatGPT avec un taux de 0.88.

Les trois LLMs affichent une précision et un score F1 relativement équilibrés, bien que plus faibles que leurs scores de rappel. Cela montre une capacité stable à identifier les vulnérabilités, mais avec un taux de faux positifs élevé, ce qui signifie qu'ils classent souvent à tort des codes bénins comme vulnérables.

4.6.2 Classification des vulnérabilités selon le type CWE

A. Performances globales selon le type du code

– **Codes vulnérables**

Sur les 364 codes vulnérables, les résultats résumés à la table 4.10 montrent que DeepSeek et Llama surpassent ChatGPT en termes d'attribution du bon type CWE lorsque le code vulnérable est correctement identifié, avec respectivement 80 et 79 types de vulnérabilités correctement attribués, contre seulement 47 pour GPT. Cependant,

les trois modèles génèrent un nombre important de faux positifs (CWE incorrectement attribués).

Lorsqu'il s'agit de codes vulnérables incorrectement classés, les LLMs tentent malgré tout d'identifier la catégorie de vulnérabilité, même s'ils considèrent le code comme étant bénin. DeepSeek, bien qu'ayant le plus grand nombre de vulnérabilités correctement classées dans cette catégorie, propose souvent des CWE incorrects, ce qui montre une tendance à des prédictions aléatoires nécessitant une analyse plus approfondie pour ces réponses correctes dans un contexte inapproprié.

TABLE 4.10 – Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans l'attribution du type CWE pour les codes vulnérables

	Vulnérable correctement classé		**Vulnérable incorrectement classé**	
	CWE Correct	**CWE Incorrect**	**CWE Correct**	**CWE Incorrect**
ChatGPT	47	283	3	31
Llama	79	265	5	15
DeepSeek	80	228	13	43

– **Codes bénins**

Sur l'ensemble des 179 codes bénins, les résultats de la table 4.11 révèlent des anomalies dans le comportement des trois LLMs. D'un côté, ils ont tendance à attribuer un type CWE à un code bénin correctement classé, bien que la plupart de leurs réponses soient incorrectes. D'un autre côté, Llama et DeepSeek parviennent parfois à proposer des CWE corrects pour des codes bénins qu'ils considèrent comme vulnérables (40 pour Llama et 36 pour DeepSeek), surpassant ChatGPT, qui n'en propose que 10. Cela suggère la nécessité d'une investigation plus approfondie sur ces cas particuliers afin de déterminer s'il existe des éléments dans la version corrigée qui guideraient le LLM à proposer le bon type CWE.

TABLE 4.11 – Comparaison des performances globales de ChatGPT avec celles de Llama et de DeepSeek dans l'attribution du type CWE pour les codes bénins

Model	**Bénin correctement classé**		**Bénin incorrectement classé**	
	CWE Correct	**CWE Incorrect**	**CWE Correct**	**CWE Incorrect**
GPT	3	13	10	153
Llama	1	8	40	130
DeepSeek	5	25	36	113

B. Performances par langauge de programmation

Pour une comparaison plus approfondie entre ChatGPT et les autres LLMs dans l'attribution des vulnérabilités à leur catégorie CWE correcte, il est pertinent d'adopter une approche granulaire en analysant leurs performances spécifiques pour chaque langage de programmation, comme rapporté dans les deux figures, 4.4 et 4.5.

Les résultats de l'expérimentation montrent que, pour les cinq langages étudiés, la majorité des prédictions de types CWE sont incorrectes pour les trois modèles, même lorsqu'ils identifient correctement un code vulnérable. Cela révèle une forte tendance aux erreurs dans l'attribution des types CWE.

Cependant, des améliorations sont observées pour certains langages, notamment Java, PHP et Python, où les modèles montrent une meilleure capacité à identifier le type CWE correct pour un code vulnérable bien classé. ChatGPT reste en troisième position derrière DeepSeek et Llama respectivement pour ces langages.

Lorsqu'un code vulnérable est classé comme non vulnérable, les cas de prédictions correctes de CWE, notamment pour ChatGPT, sont inexistants ou très faibles, surtout pour les langages C, C++ et PHP. Quelques résultats positifs sont toutefois observés pour ChatGPT en Java et Python. Llama, de son côté, n'a proposé un CWE correct que pour le langage PHP dans ce cas. DeepSeek surpasse ChatGPT et Llama pour les langages C, C++ et Java, tandis qu'il affiche des performances équivalentes à Llama pour PHP et à ChatGPT pour Python.

Pour les codes bénins correctement classés, les trois LLM ne proposent généralement pas de CWE pour la majorité des langages, à l'exception de PHP, où les trois modèles parviennent à fournir des réponses justes, bien que très faibles, et de C, où ChatGPT et DeepSeek réussissent également à en proposer. Ce comportement, observé sur des codes corrigés, mérite une investigation approfondie, notamment pour ces cas particuliers en

PHP et C. Par ailleurs, les CWE incorrectes restent globalement peu nombreuses pour les trois modèles, quel que soit le langage.

Pour le langage Java, ni Llama ni ChatGPT ne proposent de CWE (correcte ou incorrecte) pour un code bénin correctement classé. Le même comportement est observé pour Llama en Python, suggérant que ces modèles ne produisent pas de prédictions aléatoires pour ces deux langages. Ce phénomène, inhabituel, nécessite une analyse plus approfondie pour comprendre ces comportements anormaux.

De manière surprenante, quand le code est bénin mais que les trois LLM le considèrentincorrectement comme vulnérable, les trois LLM proposent parfois des CWE correctes, bien que les résultats varient selon les langages. Dans ce contexte, Llama et DeepSeek surpassent ChatGPT en PHP, Python et Java. En revanche, ChatGPT ne propose pas de CWE pour Python et Java, tandis que DeepSeek ne fournit aucune CWE pour C++. Curieusement, Llama parvient systématiquement à proposer des CWE correctes, bien que faibles, pour tous les langages dans ce cas. Cela dit, leurs prédictions restent majoritairement erronées dans cette situation.

De manière générale, les LLMs éprouvent des difficultés significatives à attribuer correctement les types CWE aux codes vulnérables, avec des performances variables selon les langages de programmation et une forte tendance aux prédictions incorrectes, particulièrement marquée dans certains cas. Pour les codes non vulnérables, les prédictions restent également limitées. Bien qu'une légère amélioration soit observée, des anomalies dans le comportement des LLMs subsistent. Ces modèles nécessitent encore des progrès considérables pour affiner leur analyse des codes exempts de vulnérabilités.

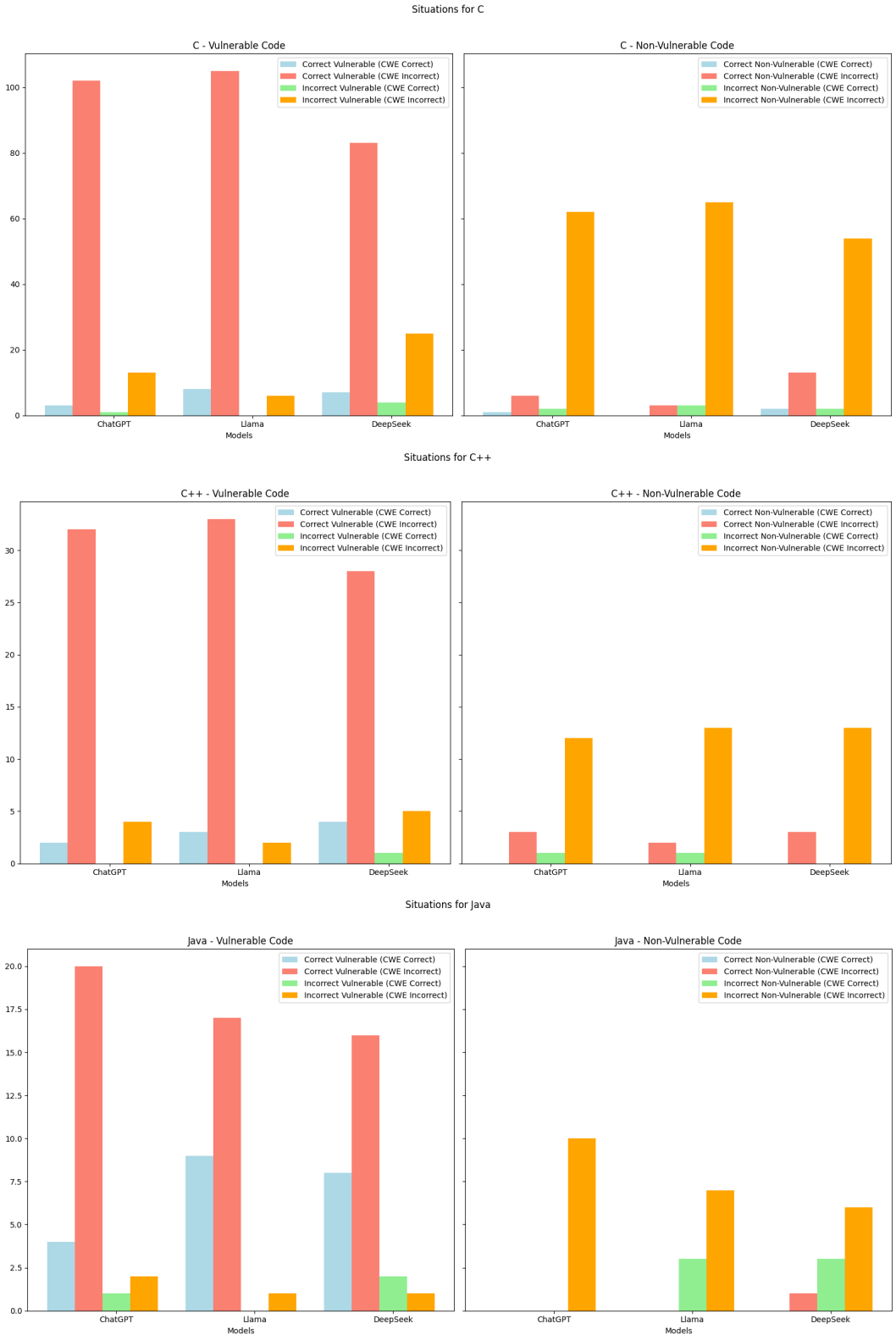

FIGURE 4.4 – Comparaison des performances de ChatGPT a celles de Llama et DeepSeek dans la classification des vulnérabilités selon le type CWE pour les langages C, C++ et Java.

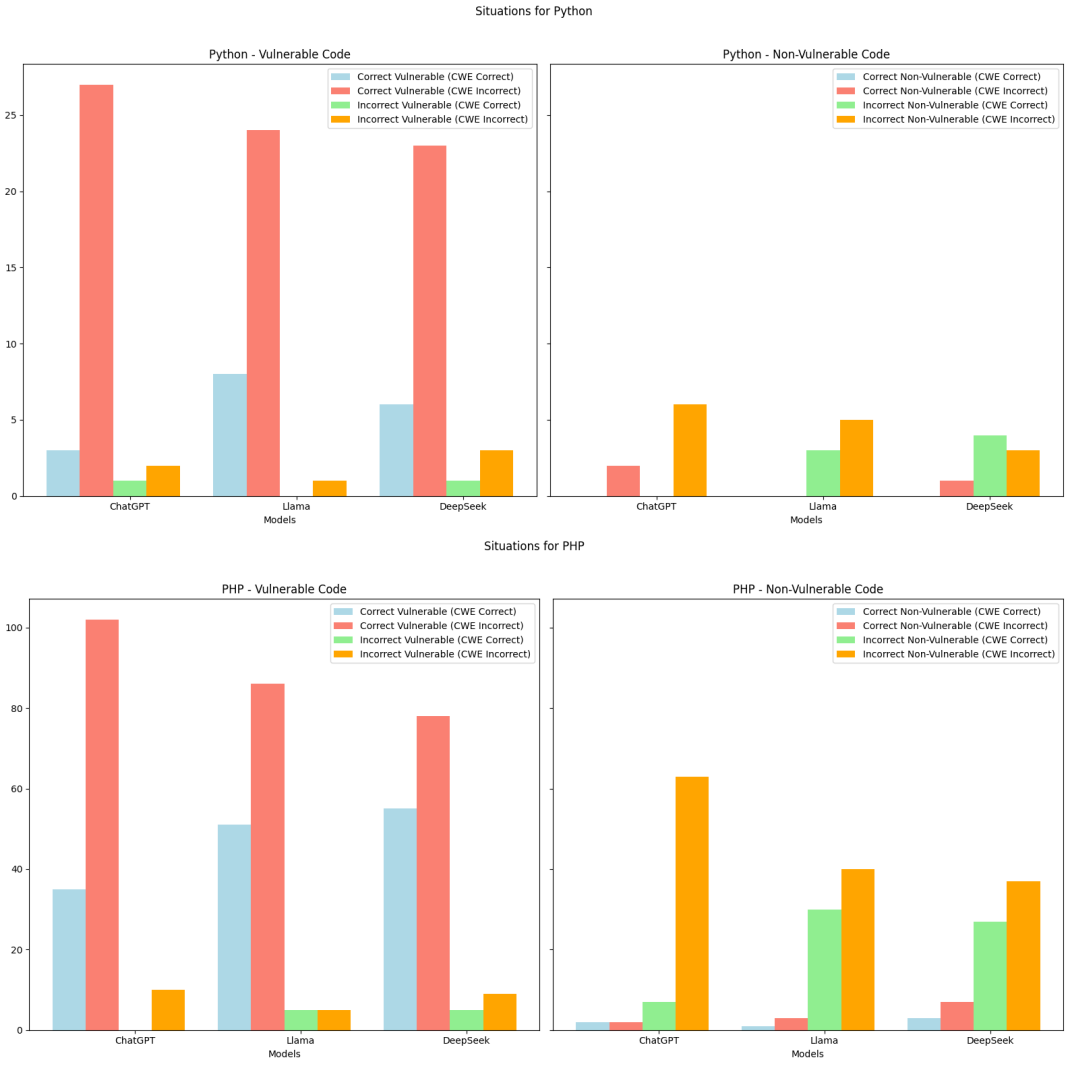

FIGURE 4.5 – Comparaison des performances de ChatGPT avec celles de Llama et DeepSeek dans la classification des vulnérabilités selon le type CWE pour les langages PHP et Python

C. Performances en fonction des CWE les plus réccurentes

Pour comparer les capacités de ChatGPT avec celles des deux autres LLMs dans l'attribution correcte des types de CWE aux vulnérabilités, nous avons également identifié les types de CWE les plus fréquents parmi les CVEs communes analysées par les trois modèles. Sur les 543 vulnérabilités traitées, nous avons sélectionné les 10 types de CWE les plus récurrents pour évaluer l'exactitude de chaque modèle sur ces catégories spécifiques.

Les types de CWE les plus fréquents et leurs occurrences respectives dans les données analysées sont les suivants : CWE-79 apparaît 110 fois, suivie de CWE-20 et CWE-119, présentes chacune 28 fois. CWE-125 est recensée 22 fois, tandis que CWE-476 et CWE-89 apparaissent chacune 20 fois. CWE-200 est observée 19 fois, CWE-787 est identifiée 18 fois, CWE-22 apparaît 17 fois, et enfin, CWE-264 est mentionnée 13 fois. La description de chaque type de CWE répertorié est donnée à la Table 4.3.

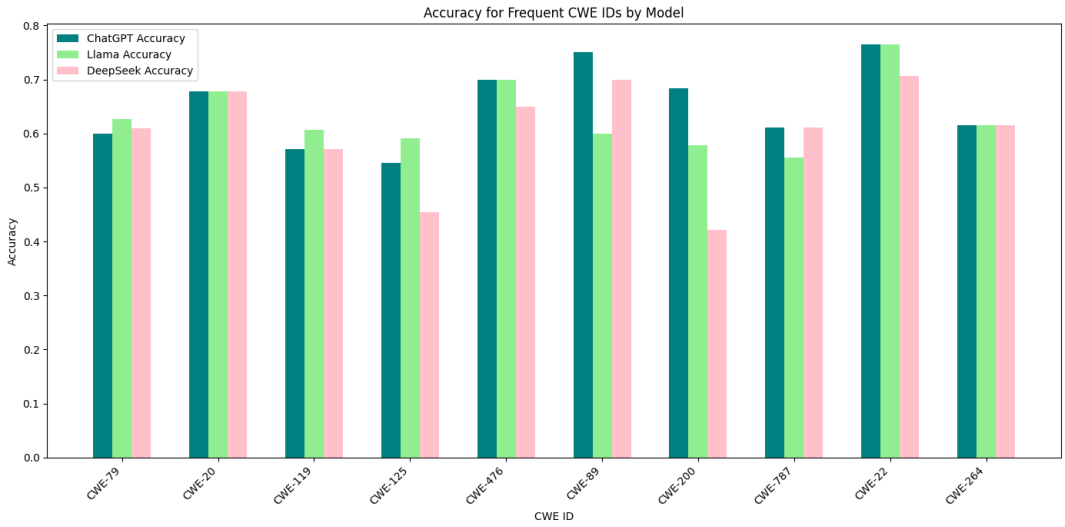

FIGURE 4.6 – Comparaison de l'exactitude de ChatGPT dans la classification des CWE récurrentes à celles de Llama et DeepSeek

Les résultats, illustrés à la Figure 4.6, montrent que les trois modèles présentent une exactitude identique pour les CWE-264 et CWE-20, reflétant des performances équivalentes sur ces vulnérabilités. ChatGPT surpasse Llama et DeepSeek pour les CWE-89 et CWE-200. En revanche, ChatGPT atteint des performances similaires à celles de Llama pour les CWE-22 et CWE-476, et fait jeu égal avec DeepSeek pour les CWE-

119 et CWE-787. Cependant, Llama dépasse ChatGPT pour les CWE-79, CWE-119 et CWE-125, tandis que DeepSeek surpasse légèrement ChatGPT uniquement pour la CWE-79. ChatGPT affiche une exactitude globale satisfaisante dans la classification des CWE récurrentes, souvent comparable à celle de Llama et DeepSeek.

Cette comparaison souligne que chaque modèle possède des points forts et des faiblesses distincts en matière d'exactitude dans la classification des vulnérabilités, ces performances variant en fonction du type de CWE considéré.

4.6.3 Localisation des lignes vulnérables

La table 4.12 compare les performances de ChatGPT, DeepSeek et Llama en termes de détection de lignes vulnérables (IoU) et de fausses alertes (IFA) pour différents langages de programmation, en distinguant les codes vulnérables et bénins.

À la lumière des valeurs d'IoU pour les codes vulnérables, il apparaît clairement que ni ChatGPT ni les autres modèles (Llama et DeepSeek) n'identifient efficacement les lignes vulnérables. Un IoU proche de 1 indiquerait une correspondance idéale entre les lignes vulnérables identifiées et celles réellement vulnérables. Or, les valeurs observées pour tous les modèles restent proches de 0 dans la plupart des cas, reflétant une incapacité générale à localiser précisément les lignes vulnérables.

Pour les codes vulnérables, ChatGPT surpasse les deux autres modèles dans certains cas, bien que ses performances demeurent insuffisantes. Par exemple, il affiche un meilleur IoU pour Python (0.0244) et C++ (0.0098). Cependant, pour d'autres langages comme PHP et Java, les valeurs d'IoU restent également proches de zéro, révélant une incapacité généralisée à identifier avec précision les lignes vulnérables. DeepSeek se distingue légèrement en C avec un IoU supérieur (0.0101), tandis que Llama atteint la meilleure performance en Java (0.0009) mais extrêmement faibles.

En termes d'IFA (fausses alertes), ChatGPT affiche des performances compétitives, notamment pour C++ et Python, où il génère moins de fausses alertes que DeepSeek. Cependant, Llama se démarque par des valeurs d'IFA minimales dans presque tous les langages, indiquant qu'il génère globalement moins de fausses alertes.

Pour les codes bénins, ChatGPT, tout comme les autres modèles, affiche des valeurs d'IoU nulles, à l'exception de DeepSeek qui obtient un faible score pour le langage C (0.00038), tout en maintenant des IFA modérées. Néanmoins, Llama reste le plus performant avec les IFA les plus faibles dans tous les langages. Cela suggère que les modèles

TABLE 4.12 – Comparaison des valeurs de l'IFA et IoU pour les 3 modèles sur les codes vulnérables et bénins, par langage de programmation.

		Vulnérable		**Bénin**	
Langage	**Modèle**	**avg_IoU**	**avg_IFA**	**avg_IoU**	**avg_IFA**
C	DeepSeek	0.0101	7.14	0.00038	8.92
	Llama	0	1.83	0	2.08
	ChatGPT	0.0011	6.08	0	5.67
Python	DeepSeek	0	8.17	0	8.57
	Llama	0.0062	1.87	0	1.75
	ChatGPT	0.0244	4.63	0	3.5
PHP	DeepSeek	0.0090	4.98	0	7.29
	Llama	0.0036	1.29	0	1.45
	ChatGPT	0.0092	4.21	0	4.75
C++	DeepSeek	0	11.5	0	4.69
	Llama	0	1.66	0	1.85
	ChatGPT	0.0098	4.11	0	2.92
Java	DeepSeek	0	6.5	0	8.55
	Llama	0.0009	1.76	0	1.2
	ChatGPT	0	4.33	0	2.6

identifient à tort des lignes vulnérables dans des codes déjà corrigés. Ces fausses alertes compliquent inutilement le travail du développeur, l'obligeant à analyser et corriger des lignes identifiées à tort comme vulnérables par le modèle, alors que le code est en réalité exempt de vulnérabilités.

Par conséquent, ChatGPT montre de meilleures performances dans des langages comme Python et C++ pour les codes vulnérables, mais il est moins efficace que Llama pour éviter les fausses alertes. DeepSeek, bien qu'affichant des performances stables dans tous les langages, est surpassé dans la plupart des cas.

4.6.4 Correction automatique de vulnérabilités

Nous effectuons une évaluation manuelle des correctifs proposés par les LLMs en suivant la méthodologie décrite dans la section 4.4. Cette approche consiste à sélectionner de manière ciblée certaines réponses des trois LLMs pour évaluer si les correctifs proposés ont un impact sur l'existence de bilans positifs. Elle se concentre sur les cas où le LLM a correctement classé le code comme vulnérable, l'a attribué à son type CWE exact, a

localisé les bonnes lignes vulnérables pour toutes les tâches de sécurité, en considérant la proposition d'une correction appropriée de la vulnérabilité comme critère déterminant.

En parallèle, nous avons identifié, pour chaque LLM, le nombre de bilans négatifs correspondant aux cas où les codes bénins sont correctement classés.

Bilan négatif. Parmi les 179 codes bénins présents dans le fichier des réponses issues des 543 CVEs communes de chaque LLM, nous avons répertorié les cas où ces codes sont correctement classés, ces derniers constituant des bilans négatifs. Les résultats obtenus sont les suivants :

- • ChatGPT : 16 cas correctement classés, dont 3 avec des CWE exactes.
- • Llama : 9 cas correctement classés, dont 1 avec une CWE exacte.
- • DeepSeek : 30 cas correctement classés, dont 5 avec des CWE exactes.

Sachant que ChatGPT a produit 22 bilans négatifs sur les données initiales, 16 d'entre eux concernent les données communes aux deux autres LLMs.

Bilan positif. Pour vérifier l'existence de bilans positifs, nous avons d'abord extrait, pour chaque LLM, les réponses où les codes vulnérables sont correctement identifiés et associés au type CWE approprié, puis sauvegardé les résultats dans trois nouveaux fichiers, un pour chaque LLM. Nous avons enregistré 47 cas pour ChatGPT, 79 pour Llama, et 80 pour DeepSeek.

Par la suite, afin de filtrer davantage les correctifs proposés et ne conserver que les cas où les lignes vulnérables sont entièrement ou partiellement localisées, nous avons calculé les valeurs d'IFA et d'IoU pour évaluer les lignes vulnérables détectées par chaque LLM, en les répartissant par langage de programmation. Les résultats obtenus sont synthétisés dans la table 4.13.

TABLE 4.13 – Résultats IoU et IFA par langage pour chaque modèle, permettant d'identifier les correctifs à évaluer

	ChatGPT		**Llama**		**DeepSeek**	
	avg_IoU	**avg_IFA**	**avg_IoU**	**avg_IFA**	**avg_IoU**	**avg_IFA**
Python	0	2	0	2.375	0	4.333
Java	0	5.75	0	2.333	0	7.5
C++	0	6	0	2.667	0	8.5
C	0	8	0	1.625	0	3
PHP	0.0031	3.343	0	1.137	0.0035	4.491

Bien que les résultats globaux soient insatisfaisants pour les trois LLMs, ces derniers fournissent néanmoins des indications utiles pour orienter le choix des correctifs à évaluer. Les valeurs d'IoU montrent que l'analyse des correctifs se concentrera principalement sur les codes PHP pour ChatGPT (0,0031) et DeepSeek (0,0035), même si ces valeurs restent extrêmement basses. En revanche, Llama affiche des valeurs nulles pour l'IoU, ce qui indique qu'il est incapable d'identifier correctement les lignes vulnérables parmi les 79 cas examinés, ne générant ainsi aucun bilan positif. Cela met en lumière ses limites dans la proposition d'une analyse complète et correcte d'un code vulnérable dans son ensemble, malgré des performances notables sur certaines tâches de sécurité pour certains langages.

À l'issue de ce filtrage, nous avons constaté que ChatGPT dispose de 35 cas en PHP, tandis que DeepSeek en compte 55. En tenant compte des valeurs très faibles de l'IoU, on peut en déduire qu'il existe probablement un très petit nombre de cas où les lignes vulnérables proposées étaient effectivement correctes. Pour cela, nous avons recalculé les valeurs d'IFA et d'IoU pour chaque entrée du fichier, au lieu de calculer la moyenne, afin d'identifier avec précision les cas où les deux modèles ont correctement détecté les lignes vulnérables. Ces cas ont ensuite été considérés pour évaluer la pertinence de leur proposition de correctif et pour décider si un bilan positif devait être attribué. Les résultats ont montré que, pour ChatGPT, sur les 35 cas, un seul a obtenu un IoU égal à 0,1111, tandis que pour DeepSeek, trois cas sur les 55 ont obtenu les résultats suivants : IoU de 0,1111, 0,0447 et 0,0384, dont un est similaire à celui de ChatGPT.

ChatGPT :

Pour le cas unique de ChatGPT, le code à évaluer présente une vulnérabilité de type CWE-79 (*Neutralisation incorrecte des entrées lors de la génération de pages Web, ou "Cross-site Scripting"*). ChatGPT a réussi à identifier une ligne vulnérable parmi celles existantes dans la référence, mais a également généré des fausses alertes en désignant à tort d'autres lignes comme vulnérables . Dans sa tentative de correction, il n'a pas neutralisé efficacement la vulnérabilité, se contentant de réécrire la ligne vulnérable sans la modifier, laissant ainsi la faille intacte.

Pour corriger cette vulnérabilité, il fallait remplacer `xss_clean($autoSize)` par int-val`($autoSiz)` afin de s'assurer que $autoSize soit traité comme un entier, réduisant ainsi le risque lié aux entrées malveillantes. Par ailleurs, une sécurisation systématique

des sorties, notamment en utilisant htmlentities pour la variable settingsType, aurait été indispensable afin de prévenir les injections XSS lors de l'affichage.

De plus, ChatGPT a tenté d'apporter des corrections, notamment en introduisant `htmlspecialchars`. Cependant, ses modifications concernaient des lignes incorrectement identifiées comme vulnérables, ce qui a entraîné des changements inutiles et altéré la structure du code. Une approche plus rigoureuse, incluant des validations précises des entrées et une sécurisation systématique des sorties, serait indispensable pour garantir une protection efficace contre ce type de menace.

Ainsi, aucun bilan positif n'a pu être attribué à ChatGPT.

DeepSeek :

Cas1 : Contenant la vulnérabilité CWE-347(*Improper Verification of Cryptographic Signature*)

Le code évalué contient deux lignes vulnérables identiques, résultant de l'absence de validation rigoureuse des signatures cryptographiques, ce qui expose le système à l'acceptation de signatures non valides ou malveillantes. DeepSeek n'a identifié qu'une seule de ces deux lignes vulnérables. En revanche, il a signalé d'autres lignes non vulnérables comme nécessitant des corrections. Les lignes proposées par DeepSeek pour correction incluent la ligne vulnérable identifiée, mais celle-ci est restée inchangée, indiquant qu'aucune correction n'a été jugée nécessaire. Par conséquent, la vulnérabilité n'a pas été corrigée. Il aurait été nécessaire de passer à `EXC_C14N_COMMENTS` pour s'assurer que tous les commentaires inclus soient soigneusement validés et exempts de contenu potentiellement malveillant.

Le reste de la proposition de correction ne représente qu'une tentative d'amélioration, mais sans efficacité réelle. Avant correction, les lignes utilisaient des pratiques et des validations explicites pour réduire les risques liés aux valeurs nulles et offrir une défense basique. Cependant, après correction, la suppression de ces mécanismes a introduit des risques supplémentaires, notamment des vulnérabilités dues à des données inattendues ou absentes, susceptibles de provoquer des erreurs de type ou des comportements imprévisibles.

DeepSeek n'a pas corrigé la vulnérabilité liée à la vérification cryptographique et, au contraire, a introduit des risques menant à d'autres vulnérabilités.

Cas2 : Contenant la vulnérabilité CWE-89(*Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*)

DeepSeek n'a réussi à identifier correctement que 3 lignes vulnérables parmi les 100 mentionnées dans les données de référence, avec un IFA de 0. Cela confirme qu'aucune fausse alerte n'a été signalée dans ce cas particulier. La première ligne qu'il a proposée était vulnérable. Cependant, plusieurs autres lignes, qui ont contribué à la présence d'une injection SQL potentielle, ont été ignorées. Sa tentative de correction ne supprime pas complètement cette vulnérabilité ; elle se concentre uniquement sur les 3 lignes proposées en explicitant la connexion active avec `$U->db_link` , qu'il inclut dans la requête SQL, contenant déjà `mysqli_real_escape_string` pour prévenir les injections SQL.

Cas3 : Contenant la vulnérabilité CWE-79 (*Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*)

Le code en question contient une seule ligne vulnérable, comme indiqué dans les données de référence. Cependant, DeepSeek a identifié 21 lignes comme potentiellement vulnérables, y compris la véritable ligne vulnérable. Ce comportement engendre un grand nombre de fausses alertes, ce qui risque d'induire les développeurs en erreur et d'alourdir leur travail d'analyse.

Concernant la correction de la véritable ligne vulnérable, DeepSeek s'en rapproche globalement sur le plan sémantique à la ligne de référence. En effet, elle respecte les principes de sécurité essentiels, tels que l'utilisation de la fonction h() pour prévenir les injections de code HTML et l'ajout d'attributs sécuritaires comme rel="noreferrer noopener". Toutefois, l'échappement excessif des caractères introduit par DeepSeek complique inutilement le code. Cette complexité pourrait entraîner des incohérences dans certains environnements. En revanche, lorsqu'il s'agit de corriger les fausses alertes, DeepSeek applique des modifications inutiles sur des lignes non vulnérables, ce qui complexifie davantage le code sans résoudre la véritable vulnérabilité.

Finalement, DeepSeek n'a pas réussi à éliminer les vulnérabilités dans les trois cas cités ci-dessus, malgré ses tentatives de correction. En conséquence, de manière similaire à Llama et ChatGPT, il n'a généré aucun bilan positif. Il est possible que ces résultats soient dus à l'utilisation de codes trop longs dans les tests.

4.6.5 Re-vérification de l'existence de vulnérabilités

La re-vérification intervient dans les situations où un code vulnérable a été mal classé ou une CWE incorrectement attribuée à un code confirmé comme vulnérable. Son objectif est de confirmer l'existence de la vulnérabilité et de localiser à nouveau les lignes vulnérables. Parmi les 543 CVEs communes, nous avons recensé les cas où une re-vérification a été effectuée sur des réponses initiales erronées pour confirmer la présence d'une vulnérabilité.

La table 4.14 ci-dessous résume les résultats obtenus pour chaque LLM, en précisant le nombre de cas soumis à re-vérification, les cas où la présence de vulnérabilités correspondant au type CWE indiqué a été confirmée par une réponse positive ("Yes") et ceux ayant reçu une réponse négative ("No"). De plus, nous avons calculé le pourcentage de réduction des prédictions aléatoires basé sur les réponses positives confirmées par "Yes".

TABLE 4.14 – Résultats de la re-vérification des vulnérabilités correspondant au type CWE indiqué pour chaque LLM

	Cas revérifiés	**Non**	**Oui**	**Taux de réduction**
ChatGPT	317	271	46	14.51%
Llama	285	0	285	100%
DeepSeek	284	196	88	30.98%

Llama atteint une réduction parfaite avec 100%, indiquant une élimination complète des prédictions aléatoires après re-vérification. DeepSeek suit avec un taux de 30,98%, ce qui reflète une performance modérée dans l'amélioration de la précision après re-vérification. ChatGPT, quant à lui, affiche un taux de 14,51%, suggérant une réduction plus limitée des prédictions aléatoires. Ces résultats montrent que Llama est nettement plus efficace dans ce contexte, tandis que ChatGPT offre une amélioration marginale.

Ces réponses doivent encore être analysées afin de valider les nouvelles lignes vulnérables identifiées.

Les résultats d'IoU illustrés à la table 4.15 montrent une faible capacité des modèles à localiser les lignes vulnérables dans le code, indiquant que les lignes proposées, reflétées par les valeurs d'IFA, ne sont en réalité que de fausses alertes. ChatGPT obtient un score de 0, indiquant une incapacité totale à identifier les lignes attendues. DeepSeek, avec un score de 0.0103, affiche une performance légèrement meilleure mais reste très limitée, tandis que Llama, avec 0.0035, présente des résultats encore plus faibles, malgré

TABLE 4.15 – Résultats IoU et IFA pour chaque modèle permettant d'évaluer les nouvelles lignes predites après la re-vérification

Modèle	**avg_IoU**	**avg_IFA**
ChatGPT	0	1.6739
DeepSeek	0.0103	3.7045
Llama	0.0035	1.1228

un taux de 100% obtenu précédemment pour la confirmation des vulnérabilités. Ces faibles scores soulignent que la localisation précise des lignes vulnérables demeure un défi pour ces modèles, même après un second tour de vérification visant à réduire les prédictions aléatoires.

À la lumière de ces expérimentations, il est clair que les LLMs ont d'énormes difficultés à repérer et corriger les vulnérabilités de sécurité, ce qui suggère qu'ils doivent encore progresser. Ces résultats pourraient être améliorés si les LLM sont entraînés graduellement, en commençant par des codes contenant une seule ligne vulnérable, puis en augmentant progressivement le nombre de lignes vulnérables, et ce, par langage et par type de vulnérabilité.

4.7 Leçons tirées de l'utilisation de trois LLMs comme réviseurs de la sécurité du code

Dans cette section, nous présentons les observations clés et pertinentes tirées de notre expérience.

4.7.1 Limite des tokens : un obstacle à l'analyse de programmes complets

La taille maximale des tokens dans les LLMs représente une contrainte majeure pour l'analyse de programmes complets. Plusieurs stratégies ont été adoptées pour surmonter cette limitation. Dans notre approche, nous avons choisi de travailler avec des codes respectant la limite de tokens la plus basse parmi les trois LLMs, à savoir celle de ChatGPT (16 358 tokens). Les stratégies mises en œuvre incluent :

- La suppression de tous les commentaires dans le code.
- La réinitialisation du contexte d'interaction avec le LLM après le traitement de chaque CVE.

4.7.2 CVEs traitées par les trois LLMs diffèrent

Malgré les ajustements effectués pour sélectionner des codes respectant la taille de token autorisée pour le contexte, la capacité des LLMs à traiter les CVEs reste limitée et varie d'un modèle à l'autre. Les CVEs effectivement analysées par chaque LLM sont répertoriées dans la table 4.7. Cela persiste même après s'être assuré que les fichiers d'entrée contiennent exactement les mêmes vulnérabilités.

Il arrive parfois que des réponses inconsistantes et détaillées, lorsque les modèles ne respectent pas les instructions données dans les prompts ou ne suivent pas le format requis pour les réponses, affectent la taille du contexte et entraînent l'élimination du traitement de certaines CVEs.

Pour garantir une comparaison équitable, notre analyse s'est concentrée sur les CVEs communes extraites des réponses hétérogènes des LLMs. Cette approche permet d'évaluer les modèles sur un pied d'égalité.

4.7.3 Divergence dans la capacité des modèles

Nous avons observé des disparités dans l'efficacité des modèles en fonction de plusieurs facteurs :

- Le langage de programmation utilisé,
- Les types spécifiques de CWE ciblés,
- Le type des programmes analysés, vulnérables et bénins.
- Les vulnérabilités vues ou non vues durant la phase d'entraînement du LLM, comme c'est le cas pour ChatGPT, restent à vérifier pour les deux autres LLMs, DeepSeek et Llama, dans le futur.

Ces différences méritent une étude approfondie afin de mieux comprendre les points forts et les limites des LLMs selon ces variables.

4.7.4 Incomplétude des réponses générées par les LLMs

Lors de l'analyse des LLMs, il a été observé que certaines réponses étaient incomplètes ou tronquées, tant pour les lignes vulnérables prédites que pour les corrections proposées. Ce phénomène a été particulièrement fréquent chez DeepSeek et Llama. Ce comportement est probablement dû à une combinaison de limitations techniques ainsi qu'aux spécificités de chaque modèle, telles que leur architecture et leurs données d'entraînement. Une analyse plus poussée des paramètres et de l'entraînement des modèles pourrait permettre de mieux comprendre ces problèmes et d'y remédier.

4.7.5 Incohérence des réponses : un obstacle à l'évaluation automatisée des LLMs

L'un des principaux obstacles rencontrés réside dans l'incapacité d'automatiser complètement le processus. Les réponses générées par les LLMs présentent fréquemment des incohérences ou deviennent verbeuses en ajoutant des explications inutiles, notamment lorsque les modèles ne respectent pas les instructions des prompts ou dévient du format attendu. Cela impose une intervention humaine régulière pour prétraiter et adapter les réponses aux processus d'évaluation, afin d'assurer une interprétation correcte des résultats.

4.7.6 Anomalies dans les réponses des LLMs

- Les LLMs, notamment Llama et ChatGPT, ont parfois répondu "None" à des demandes de correction, malgré l'identification correcte de vulnérabilités et la proposition de lignes vulnérables.
- Llama et DeepSeek ont tendance à proposer des lignes vulnérables pour des codes bénins correctement classés comme non vulnérables. Ce type d'anomalie n'a pas été constaté avec ChatGPT.
- Les trois modèles sont enclins à proposer des lignes comme vulnérables dans des codes déjà corrigés. Ce comportement aléatoire identifie à tort des lignes bénignes comme vulnérables, augmentant ainsi les fausses alertes et incitant à des corrections inutiles.
- Les trois LLMs montrent une tendance à attribuer des types CWE à des codes bénins correctement classés, bien que la majorité de ces attributions soient erronées. Cependant, ils parviennent parfois à identifier avec précision des types CWE pour des codes bénins qu'ils évaluent à tort comme vulnérables. Cette situation met en évidence la nécessité d'une analyse approfondie afin de comprendre si des éléments spécifiques dans les versions corrigées influencent les LLMs dans leur classification des types CWE.

Chapitre 5

Conclusion

Dans le cadre de ce mémoire, nous avons évalué l'efficacité de ChatGPT pour l'analyse de la sécurité de codes issus de projets logiciels réels, en mettant en évidence ses forces et ses limites, tant avant qu'après sa date d'entraînement, et en le comparant à deux autres LLMs : DeepSeek et Llama. Nous avons implémenté une méthodologie basée sur un processus en plusieurs étapes, comprenant la collecte de données pertinentes provenant de projets réels, la rédaction de prompts adaptés, la création d'un dialogue interactif avec le modèle, ainsi que la sauvegarde et l'évaluation des réponses du LLM.

D'une part, les résultats révèlent que les performances de ChatGPT varient en fonction des données rencontrées ou non lors de sa phase d'entraînement. Bien que ChatGPT ait démontré une bonne capacité à classer correctement les codes vulnérables, il a également généré un nombre important de faux positifs. Le modèle éprouve des difficultés à identifier les codes bénins, échouant dans 94,13% des cas. Après septembre 2021, certaines améliorations sont observées pour des langages comme C et C++, avec des précisions atteignant 0,9091, bien que ces résultats contrastent avec d'autres études qui ont rapporté des performances plus faibles. Toutefois, l'attribution correcte des types de CWE reste variable : ChatGPT atteint une bonne exactitude pour certains types, comme CWE-120 (0,8947) et CWE-400 (0,8333), mais rencontre des difficultés pour les vulnérabilités ayant fréquence élevée dans les données de test telles que CWE-79 et CWE-119. L'analyse des correctifs a également révélé des limites importantes. ChatGPT tend à ne pas proposer de véritables corrections pour les lignes qu'il identifie comme vulnérables, en particulier lorsqu'il s'agit de codes bénins qu'il a classés comme vulnérables. Dans certains cas, le modèle a répondu par (None) pour les codes bénins laissant certaines vulnérabilités sans solution pertinente, soulignant ainsi une incohérence dans ses

réponses. De plus, aucun bilan positif n'a été enregistré, et ChatGPT n'a pas réussi à fournir une analyse complète et correcte des codes vulnérables. Enfin, la re-vérification, bien qu'elle ait été tentée pour améliorer les prédictions, n'a pas réussi à réduire la prédiction aléatoire dans 89,24% des cas, démontrant ainsi les limites actuelles du modèle pour une analyse de sécurité fiable.

D'autre part, ses perfornaces sont variables en comparaison avec celles des deux autres LLMs. En termes de classification binaire, Llama surpasse ChatGPT et DeepSeek avec un rappel global d'environ 94%, tandis que ChatGPT atteint 90% et DeepSeek 84%. Llama reste supérieur à ChatGPT et DeepSeek en termes de rappel dans la majorité des langages. Une exception notable est PHP, où ChatGPT et Llama affichent un rappel identique de 0,93, tandis que DeepSeek suit avec un rappel de 0,90, indiquant que les trois modèles excellent dans la détection des vulnérabilités en PHP. En revanche, DeepSeek, en Java, égalise le rappel de ChatGPT avec un taux de 0,88. Parmi les 364 codes vulnérables communs, DeepSeek et Llama surpassent ChatGPT en attribuant correctement les types CWEs, avec respectivement 80 et 79 attributions correctes, contre 47 pour ChatGPT. Toutefois, tous les modèles génèrent de nombreux faux positifs. Les trois modèles ont une exactitude identique pour les CWE-264 et CWE-20. ChatGPT surpasse Llama et DeepSeek pour les CWE-89 et CWE-200. Ses performances sont équivalentes à celles de Llama pour les CWE-22 et CWE-476, et à celles de DeepSeek pour les CWE-119 et CWE-787. Llama dépasse ChatGPT pour les CWE-79, CWE-119 et CWE-125, tandis que DeepSeek le surpasse légèrement pour la CWE-79. Concernant la localisation des lignes vulnérables, qui constitue un défi pour les trois LLMs, ChatGPT excelle en Python et C++ pour les codes vulnérables, malgré des valeurs IoU faibles, mais est moins efficace que Llama pour éviter les fausses alertes. DeepSeek présente des performances stables mais est surpassé dans la plupart des cas. Aucun des modèles n'a réussi à corriger les vulnérabilités présentes dans les codes sélectionnés pour l'évaluation. Ainsi, tout comme Llama et ChatGPT, DeepSeek n'a généré aucun bilan positif. Ces résultats pourraient être dus à l'utilisation de codes trop longs lors des tests.

Ces résultats soulignent la nécessité de poursuivre les recherches pour affiner les capacités des LLMs, en particulier ChatGPT, dans le domaine de la sécurité du code. Il est essentiel de réduire les faux positifs et d'améliorer l'exactitude des prédictions de CWEs et les lignes vulnérables pour rendre les modèle plus fiables et efficaces dans les tâches de sécurité. Actuellement, ChatGPT et les deux autres LLms ne peuvent pas être considérés comme des outils d'analyse de sécurité sur lesquels on peut compter, et leur

intégration dans les environnements de développement n'est pas envisageable dans leur état actuel. Au contraire, il pourrait compliquer les tâches des développeurs et mettre leur code en péril en offrant des alertes et des recommandations de sécurité inexactes.

Contraintes. Plusieurs contraintes ont émergé durant l'expérimentation et l'interprétation des résultats. Au début, nous avions envisagé d'inclure les langages R et Matlab dans notre analyse, mais leur taille dépassait la limite de tokens de ChatGPT, ce qui nous a contraints à nous limiter aux langages C, C++, PHP, Java, et Python. Parallèlement, les réponses détaillées et élaborées du LLM, lorsqu'il ne respecte pas la consigne, peuvent affecter la limite de tokens réservée aux réponses, réduisant ainsi la taille du contexte et empêchant le traitement complet de certaines CVEs. Certaines CVEs posent des problèmes lors de l'exécution, alors que dans d'autres situations, les mêmes CVEs ne causent aucun problème.

Lorsque nous avons commencé à appliquer les métriques, Distance de Levenshtein, IFA et IoU pour évaluer les lignes vulnérables proposées par ChatGPT, cela s'est révélé inefficace en raison de la variabilité des réponses de ChatGPT, qui alternent entre numéros de lignes et contenu textuel, même avec la spécification d'un prompt initial demandant une réponse au format JSON. Comme diff_parsed combine ces deux éléments, il est impossible de comparer efficacement sans texte correspondant des deux côtés. De plus, les numéros de lignes fournis par GPT ne correspondent souvent pas à ceux de diff_parsed, rendant l'application de ces métriques complexe et peu fiable. Nous avons donc modifié le prompt pour guider ChatGPT à fournir sa réponse sous forme de dictionnaire Python, incluant à la fois les numéros de lignes et le contenu des lignes. Par la suite, nous avons préféré utiliser uniquement les métriques IFA et IoU.

Futurs travaux. Comme perspective, une piste prometteuse serait de concentrer les futures analyses sur les réponses correctes fournies par les LLMs. Une investigation approfondie de ces cas permettrait d'identifier des éléments clés et des schémas récurrents susceptibles d'améliorer leurs performances. Ces travaux offriraient non seulement des opportunités d'optimisation pour ces modèles, mais également une meilleure évaluation de leur efficacité, tout en renforçant leur utilisation dans des contextes de sécurité logicielle.

Bibliographie

[1] https://platform.openai.com/tokenizer. [Accessed 11-07-2024].

[2] https://platform.openai.com/. [Accessed 11-08-2024].

[3] https://platform.openai.com/docs/models/gpt-3-5-turbo/. [Accessed 11-08-2024].

[4] CVE-SearchCVEList(mitre.org). [Accessed 11-08-2024].

[5] https://nvd.nist.gov/. [Accessed 11-08-2024].

[6] https://platform.openai.com/docs/. [Accessed 11-08-2024].

[7] Application Security Testing Tool | Software Security Testing Solutions | Checkmarx — checkmarx.com. https://www.checkmarx.com/. [Accessed 31-03-2024].

[8] DB Browser for SQLite — sqlitebrowser.org. https://sqlitebrowser.org/. [Accessed 11-08-2024].

[9] deepseek-ai/DeepSeek-V2.5 · Hugging Face — huggingface.co. https://huggingface.co/deepseek-ai/DeepSeek-V2.5. [Accessed 18-11-2024].

[10] File :Unraveling AI Complexity - A Comparative View of AI, Machine Learning, Deep Learning, and Generative AI.jpg - Wikimedia Commons — commons.wikimedia.org. https://commons.wikimedia.org/wiki/File:Unraveling_AI_Complexity_-_A_Comparative_View_of_AI,_Machine_Learning,_Deep_Learning,_and_Generative_AI.jpg. [Accessed 13-11-2024].

[11] Flawfinder Home Page — dwheeler.com. https://dwheeler.com/flawfinder/. [Accessed 01-04-2024].

[12] Google Code Archive - Long-term storage for Google Code Project Hosting. — code.google.com. https://code.google.com/archive/p/. [Accessed 31-03-2024].

[13] meta-llama/Llama-3.1-405B · Hugging Face — huggingface.co. https://huggingface.co/meta-llama/Llama-3.1-405B. [Accessed 18-11-2024].

[14] Qu'est-ce que l'apprentissage non supervisé ? | IBM — ibm.com. https://www.ibm.com/fr-fr/topics/unsupervised-learning. [Accessed 14-11-2024].

[15] Qu'est-ce qu'un chatbot à intelligence artificielle (IA) ? IA School — intelligence-artificielle-school.com. https://www.intelligence-artificielle-school.com/ecole/technologies/chatbot-ia/. [Accessed 11-08-2024].

[16] Semgrep — Find bugs and enforce code standards — semgrep.dev. https://semgrep.dev/. [Accessed 31-03-2024].

[17] Software Vulnerability - Glossary | CSRC — csrc.nist.gov. https://csrc.nist.gov/glossary/term/software_vulnerability. [Accessed 11-07-2024].

[18] Sonarqube. https://rules.sonarsource.com,. [Accessed 24-03-2024].

[19] Welcome to Bandit — Bandit documentation — bandit.readthedocs.io. https://bandit.readthedocs.io/en/latest/. [Accessed 31-03-2024].

[20] What Are Smart Contracts on Blockchain ? | IBM — ibm.com. https://www.ibm.com/topics/smart-contracts. [Accessed 14-08-2024].

[21] Ahmad, B., Tan, B., Karri, R., and Pearce, H. Flag : Finding line anomalies (in code) with generative ai. *arXiv preprint arXiv :2306.12643* (2023).

[22] Bakhshandeh, A., Keramatfar, A., Norouzi, A., and Chekidehkhoun, M. M. Using chatgpt as a static application security testing tool. *arXiv preprint arXiv :2308.14434* (2023).

[23] Bhandari, G., Naseer, A., and Moonen, L. Cvefixes : automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering* (2021), pp. 30–39.

[24] Buscemi, A. A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *arXiv preprint arXiv :2308.04477* (2023).

[25] Cheshkov, A., Zadorozhny, P., and Levichev, R. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv :2304.07232* (2023).

[26] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), pp. 508–512.

[27] Fu, M., Tantithamthavorn, C., Nguyen, V., and Le, T. Chatgpt for vulnerability detection, classification, and repair : How far are we ? *arXiv preprint arXiv :2310.09810* (2023).

[28] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. *Advances in neural information processing systems 27* (2014).

[29] Gupta, P., Ding, B., Guan, C., and Ding, D. Generative ai : A systematic review using topic modelling techniques. *Data and Information Management* (2024), 100066.

[30] Hu, S., Huang, T., İlhan, F., Tekin, S. F., and Liu, L. Large language model-powered smart contract vulnerability detection : New perspectives. *arXiv preprint arXiv :2310.01152* (2023).

[31] Keele, S., et al. Guidelines for performing systematic literature reviews in software engineering. Tech. rep., Technical report, ver. 2.3 ebse technical report. ebse, 2007.

[32] Khoury, R. *La sécurité logicielle : une approche défensive*. Presses de l'Université Laval, 2021.

[33] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker : A deep learning-based system for vulnerability detection. *arXiv preprint arXiv :1801.01681* (2018).

[34] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. Pre-train, prompt, and predict : A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys 55*, 9 (2023), 1–35.

[35] Micheelsen, S., and Thalmann, B. A static analysis tool for detecting security vulnerabilities in python web applications. *Aalborg University, Aalborg University, 31st May* (2016).

[36] Mitchell, T. M., and Learning, M. Mcgraw-hill science. *Engineering/Math 1* (1997), 27.

[37] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen : An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv :2203.13474* (2022).

[38] ORTU, M., IBBA, G., CONVERSANO, C., TONELLI, R., AND DESTEFANIS, G. Identifying and fixing vulnerable patterns in ethereum smart contracts : A comparative study of fine-tuning and prompt engineering using large language models. *Available at SSRN 4530467*.

[39] OUYANG, L., WU, J., JIANG, X., ALMEIDA, D., WAINWRIGHT, C., MISHKIN, P., ZHANG, C., AGARWAL, S., SLAMA, K., RAY, A., ET AL. Training language models to follow instructions with human feedback. *Advances in neural information processing systems 35* (2022), 27730–27744.

[40] PETERSEN, K., VAKKALANKA, S., AND KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering : An update. *Information and software technology 64* (2015), 1–18.

[41] PURBA, M. D., GHOSH, A., RADFORD, B. J., AND CHU, B. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2023), IEEE, pp. 112–119.

[42] SIDDIQ, M. L., AND SANTOS, J. C. Securityeval dataset : mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (2022), pp. 29–33.

[43] SIDDIQ, M. L., AND SANTOS, J. C. S. Generate and pray : Using sallms to evaluate the security of llm generated code. *ArXiv abs/2311.00889* (2023).

[44] SUN, Y., WU, D., XUE, Y., LIU, H., WANG, H., XU, Z., XIE, X., AND LIU, Y. When gpt meets program analysis : Towards intelligent detection of smart contract logic vulnerabilities in gptscan. arxiv 2023. *arXiv preprint arXiv :2308.03314*.

[45] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning : An introduction*. MIT press, 2018.

[46] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[47] ZHANG, C., LIU, H., ZENG, J., YANG, K., LI, Y., AND LI, H. Prompt-enhanced software vulnerability detection using chatgpt. *arXiv preprint arXiv :2308.12697* (2023).