INSA
CENTRE VAL DE LOIRE

Institut National des Sciences Appliquées

Transformer-based Vulnerability Detection in Code at EditTime : Zero-shot, Few-shot, or Fine-tuning ?

Étude Bibliographique

Auteurs de l'article :
Aaron Chan
Anant Kharkar
Roshanak Zilouchian Moghaddam
Yevhen Mohylevskyy
Alec Helyar
Eslam Kamal
Mohamed Elkamhawy
Neel Sundaresan

Auteurs de l'étude :
Mohamed Mokrani
Lamiaa Benejma
Mouna El Arraf
Thomas Aubin

Responsable du module :
Pascal Berthomé
Relecteurs :
Prénom Nom
Prénom Nom

Si la détection de vulnérabilités logicielles est désormais universellement assistée par IA et en particulier par l'utilisation de LLM, ces derniers présentent une marge de progression importante lors des phases de développement. Nous étudions en quoi les approches proposées par l'article sont novatrices

Mots-clés : Transformeurs, Vulnerabilités logicielles, Détection de vulnérabilités

28 mars 2025

Résumé

Table des matières

Table des figures

Liste des tableaux

Introduction

Première partie

Contexte et problématique

Chapitre I

Présentation du domaine et des enjeux en cybersécurité

I.1

I.1.1

I.1.1.1

$$a + b = c \tag{I.1}$$

Figure I.1 – Exemple de figure

```
print("This line will be printéd.")
print("Another line to print."
```

Listing I.1 – Un code Python

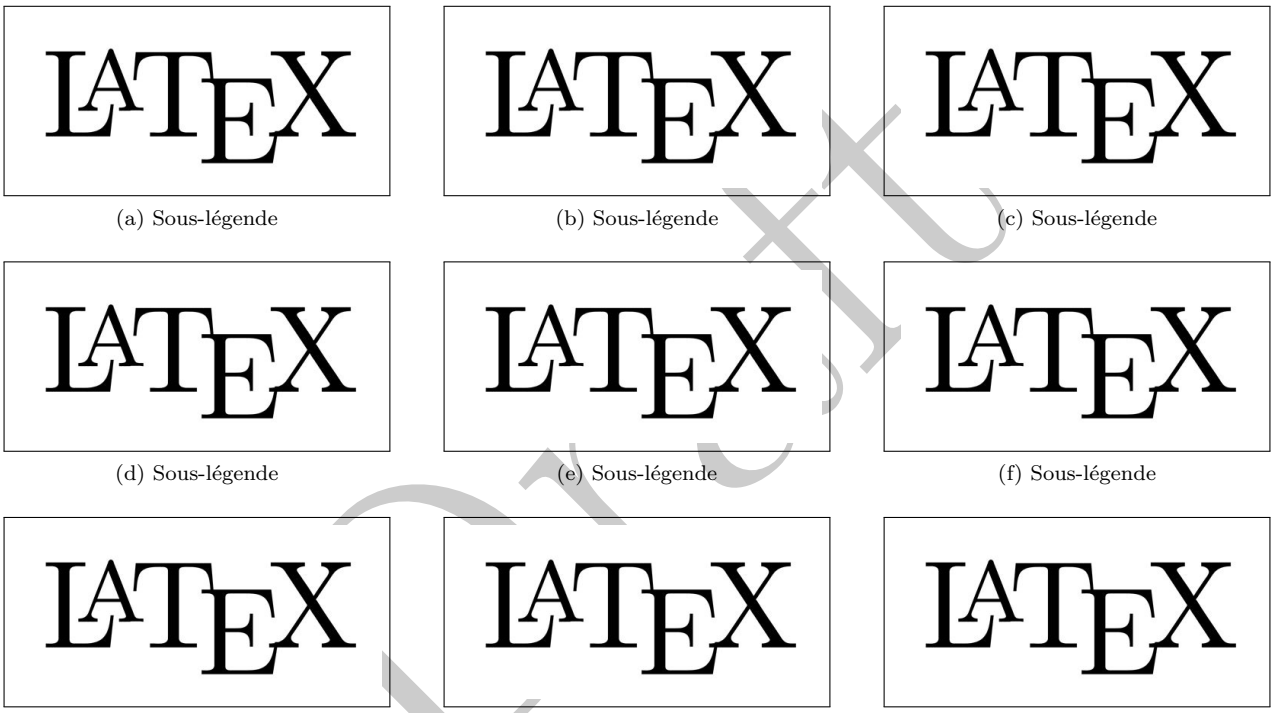
Ceci est un exemple d'encadré. Il sert à mettre en évidence des parties importantes du rapport

Donnée

TABLE I.1 – Exemple de tableau

Tâche				
Donnée			0	0

TABLE I.2 – Exemple de tableau coloré



(a) Sous-légende (b) Sous-légende (c) Sous-légende

(d) Sous-légende (e) Sous-légende (f) Sous-légende

(g) Sous-légende (h) Sous-légende (i) Sous-légende

FIGURE I.2 – Exemple avec plusieurs figures

Chapitre II

Importance de la détection des vulnérabilités

II.1

II.1.1

II.1.1.1

II.2

some text

Chapitre III

Problèmes des méthodes classiques et défis posés par la détection en temps réel

III.1

III.1.1

III.1.1.1

Deuxième partie

Apports scientifiques de l'article

L'article «Transformer-based Vulnerability Detection in Code at EditTime : Zero-shot, Few-shot, or Fine-tuning ?» explore l'utilisation des modèles de langage basés sur les transformeurs pour détecter les vulnérabilités dans le code informatique en temps réel. Cette approche vise à identifier et corriger les failles dès la phase de rédaction du code, avant même qu'il ne soit exécuté ou compilé.

Pour y parvenir, les auteurs de l'étude ont testé trois stratégies d'apprentissage : le **zero-shot learning**, le **few-shot learning** et le **fine-tuning**. Ces trois approches exploitent des modèles pré-entraînés sur des corpus de code, mais diffèrent en termes de niveau d'adaptation aux tâches spécifiques de détection de vulnérabilités.

En complément de cette étude, plusieurs modèles de langage spécialisés dans le code ont été évalués, notamment **CodeBERT, Code-Davinci-002 et Text-Davinci-003**. Les résultats des expériences menées permettent de comparer ces approches et de mesurer leur efficacité en termes de précision, de rappel et d'adaptabilité aux divers scénarios rencontrés en programmation.

Chapitre IV

Explication des trois approches (Zero-shot, Few-shot, Fine-tuning)

IV.1 Zero-shot Learning : l'application immédiate des modèles pré-entraînés

L'approche **zero-shot learning** consiste à utiliser un modèle de langage déjà entraîné sur une large base de code, sans lui fournir d'exemples spécifiques de vulnérabilités. L'objectif est de voir dans quelle mesure ce modèle est capable d'identifier des failles uniquement grâce aux connaissances acquises lors de son entraînement initial.

Cette méthode présente un avantage majeur : elle ne nécessite aucun travail d'adaptation du modèle, ce qui permet une implémentation rapide. Toutefois, cette absence de spécialisation a aussi un inconvénient majeur : la performance de détection reste limitée, avec un taux relativement élevé de **faux positifs et faux négatifs**. Le modèle peut identifier certaines failles évidentes, mais il a du mal à reconnaître des vulnérabilités plus subtiles ou spécifiques à un contexte particulier.

L'étude a montré que l'utilisation de **Text-Davinci-003** en zero-shot permet d'atteindre un **rappel de 78%**, c'est-à-dire que la plupart des vulnérabilités sont détectées. Cependant, la précision est relativement faible, ce qui signifie que le modèle génère un grand nombre d'alertes non pertinentes.

IV.2 Few-shot Learning : l'amélioration progressive grâce à des exemples ciblés

Dans l'approche **few-shot learning**, on fournit au modèle quelques exemples annotés de code vulnérable et de code sécurisé. Ces exemples lui servent de référence pour ajuster ses prédictions et améliorer sa capacité à détecter les vulnérabilités dans d'autres extraits de code.

Cette méthode présente un bon compromis entre le zero-shot et le fine-tuning. En effet, elle améliore la performance du modèle sans nécessiter un réentraînement complet. Grâce aux exemples fournis, le modèle apprend à mieux distinguer les structures de code potentiellement dangereuses.

L'expérimentation réalisée dans l'article montre que **Code-Davinci-002**, utilisé en few-shot, améliore la détection des vulnérabilités par rapport au zero-shot. Le modèle parvient à mieux contextualiser les failles et réduit le nombre de fausses alertes. Toutefois, la performance reste inférieure à celle du fine-tuning, car le modèle ne bénéficie pas d'un apprentissage approfondi sur un large jeu de données spécifique.

IV.3 Fine-tuning : l'adaptation complète à la détection des vulnérabilités

Le **fine-tuning** consiste à prendre un modèle de langage pré-entraîné et à le réentraîner sur un jeu de données spécifique contenant des exemples annotés de vulnérabilités. Cette méthode permet d'adapter entièrement le modèle à la tâche de détection des failles de sécurité.

Le principal avantage du fine-tuning est qu'il offre une **précision bien plus élevée** que les deux autres approches. En entraînant le modèle sur des données spécifiques aux vulnérabilités, on lui apprend à reconnaître avec plus de fiabilité les failles dans le code.

L'article a testé **CodeBERT** en fine-tuning sur un corpus de **500 000 extraits de code**, comprenant des exemples de vulnérabilités et de bonnes pratiques en programmation. Les résultats obtenus montrent que cette approche offre **le meilleur équilibre entre précision (59%) et rappel (63%)**. Cela signifie que le modèle détecte un grand nombre de vulnérabilités tout en limitant les fausses alertes.

Cependant, cette méthode présente aussi quelques inconvénients. Le processus de fine-tuning est coûteux en ressources computationnelles et nécessite un jeu de données annoté de grande qualité. De plus, un modèle fine-tuné sur un langage ou un type de vulnérabilité particulier pourrait être moins performant sur d'autres langages ou contextes de programmation.

Chapitre V

Présentation des modèles utilisés (CodeBERT, Code-Davinci-002, Text-Davinci-003)

L'étude compare les performances de trois modèles de langage spécialisés dans le traitement du code source.

V.1 CodeBERT : un modèle optimisé pour la compréhension du code

Développé par Microsoft et Hugging Face, **CodeBERT** est une extension de BERT spécialement entraînée sur des bases de code source. Il prend en charge plusieurs langages de programmation, dont Python, Java, JavaScript et C++. Son entraînement repose sur un large corpus de **GitHub**, ce qui lui permet d'exceller dans la compréhension syntaxique et sémantique du code.

Dans cette étude, CodeBERT a été testé en fine-tuning et a obtenu **les meilleures performances globales** en termes de précision et de rappel.

V.2 Code-Davinci-002 : un modèle génératif appliqué à la détection

Issu des modèles GPT-3 d'OpenAI, **Code-Davinci-002** est une version optimisée pour la génération et l'analyse de code. Il a été testé en **zero-shot et few-shot**, avec des résultats encourageants mais inférieurs à ceux de CodeBERT en fine-tuning.

V.3 Text-Davinci-003 : une capacité d'analyse avancée mais limitée

Text-Davinci-003, une version avancée de GPT-3, a montré de bons résultats en zero-shot grâce à son **rappel élevé**. Cependant, sa faible précision limite son utilisation pour une détection fiable des vulnérabilités.

Chapitre VI

Expérimentations et résultats observés

L'étude a comparé les modèles et les approches selon des critères de précision et de rappel.

Les résultats montrent que **CodeBERT fine-tuné** est la solution la plus efficace pour la détection des vulnérabilités en temps réel.

De plus, lorsqu'il est intégré dans **VSCode**, l'outil a permis une **réduction de 80% des vulnérabilités détectées** pendant l'édition du code, et jusqu'à **90% pour du code généré automatiquement par GitHub Copilot**.

Approche	Modèle	Précision	Rappel	Observations
Zero-shot	Text-Davinci-003	50	78	Nombre élevé de faux positifs.
Few-shot	Code-Davinci-002	55	70	Meilleure contextualisation des vulnérabilités.
Fine-tuning	CodeBERT	59	63	Meilleur équilibre entre détection et précision.

Table VI.1

Troisième partie

Impacts et applications

Chapitre VII

Améliorations du développement logiciel

VII.1 Des outils de détection classique : quel point de départ ?

VII.1.1 Détection de vulnérabilités par IA : un bref état de l'art

VII.1.1.1 Copilot : brève analyse de l'existant

VII.1.1.2 Tabnine : un exemple de standard industriel pour la sécurisation de code par IA

VII.1.2 Cas particulier du code généré par des LLM

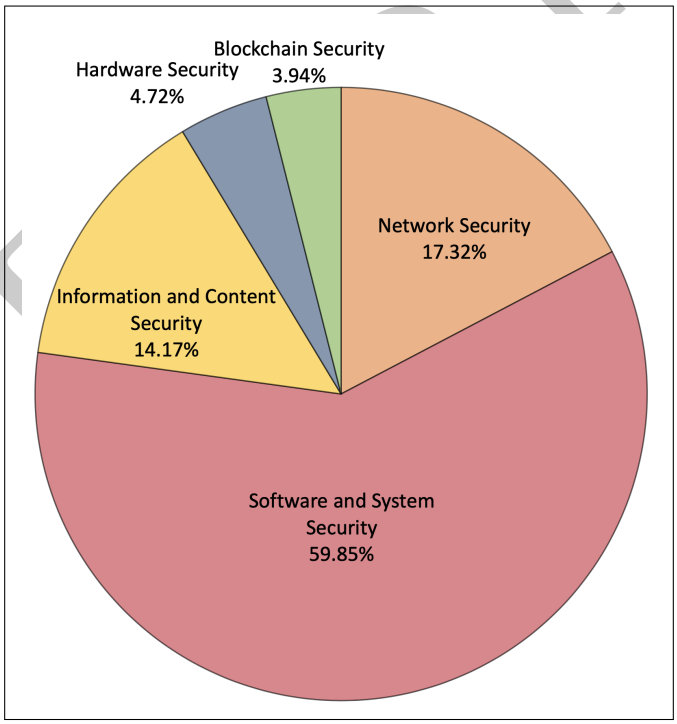


FIGURE VII.1 – Distribution of LLM usages in security domains [17]

VII.1.2.1 Pratiques actuelles de développement par LLM : exemple d'IntelliCode

VII.2 Correction et complétion pendant la phase de développement : promesses et difficultés rencontrées

VII.3 Interprétation des métriques de classification présentées

Chapitre VIII

Études de cas et intégration dans un IDE

VIII.1 Résultats préliminaires avec un déploiement des modèles sur VSCode

VIII.1.1 Méthodologie inhérente au déploiement

VIII.1.2 Résultats obtenus

VIII.1.3 Cas de figure non ou partiellement couverts par l'étude

Chapitre IX

Conséquences pour l'industrie et la recherche en cybersécurité :

IX.1

IX.1.1

IX.1.1.1

IX.2

IX.3 Continuité de la recherche

IX.3.1 Sélection et préparation des données : des méthodes encourageantes

Quatrième partie

Analyse critique et perspectives

Chapitre X

Problèmes éthiques et limites des modèles d'IA

X.1 Analyse critique et perspectives

L'article *"Transformer-based Vulnerability Detection in Code at Edit-Time : Zero-shot, Few-shot, or Fine-tuning ?"* aborde un sujet crucial dans le domaine de la cybersécurité et du développement logiciel. L'utilisation de modèles de type Transformer pour détecter les vulnérabilités en temps réel, avant la compilation, représente une avancée significative par rapport aux méthodes traditionnelles qui nécessitent une analyse post-compilation.

L'intégration de modèles d'IA dans la détection automatique des vulnérabilités soulève plusieurs questions éthiques :

X.1.1 Biais et équité des modèles

L'un des principaux problèmes des modèles basés sur l'intelligence artificielle réside dans leur dépendance aux données d'entraînement. Si les ensembles de données utilisés pour entraîner les modèles contiennent des biais (par exemple, un sous-représentation de certains langages de programmation ou types de vulnérabilités), cela peut entraîner des erreurs systématiques dans la détection.

Par ailleurs, il est possible que le modèle identifie certains types de code comme étant plus susceptibles de contenir des vulnérabilités, même si ce n'est pas le cas, simplement parce qu'ils sont statistiquement plus fréquents dans les données d'entraînement. Ce phénomène peut conduire à une stigmatisation involontaire de certains styles de programmation ou à des faux positifs pénalisants pour les développeurs.

X.1.2 Confidentialité et sécurité des données

L'intégration d'outils d'IA dans les environnements de développement pose également des questions de confidentialité. Si l'analyse du code s'effectue localement, le risque est limité. Cependant, certains outils exploitent des modèles hébergés sur des serveurs distants, ce qui implique l'envoi de fragments de code vers des infrastructures externes.

Cela soulève plusieurs préoccupations :

- **Fuites de données** : Des informations sensibles pourraient être exposées si les communications ne sont pas suffisamment sécurisées.
- **Propriété intellectuelle** : L'envoi de code source vers des serveurs tiers pourrait compromettre la protection des droits de propriété des entreprises.
- **Conformité aux réglementations** : Certaines entreprises sont soumises à des réglementations strictes en matière de gestion des données et ne peuvent pas utiliser des outils basés sur le cloud sans garanties suffisantes.

X.1.3 Responsabilité en cas d'erreur

Si un modèle d'IA échoue à détecter une vulnérabilité critique, qui en est responsable ? Cette question reste un point de débat majeur. Plusieurs scénarios sont envisageables :

- **Le développeur** : Doit-il vérifier systématiquement toutes les alertes et ne pas se fier uniquement aux recommandations du modèle ?
- **L'éditeur du logiciel** : Peut-il être tenu pour responsable s'il intègre un outil de détection automatisée qui s'avère imparfait ?

— **Le fournisseur de l'outil d'IA** : A-t-il une responsabilité légale si son modèle ne fonctionne pas correctement ?

En l'absence d'un cadre réglementaire clair, cette problématique demeure ouverte et pourrait devenir un enjeu juridique majeur à l'avenir

Chapitre XI

Évaluation du protocole de recherche

L'étude repose sur une comparaison des performances entre différentes stratégies d'apprentissage (zero-shot, few-shot et fine-tuning). L'utilisation de benchmarks standards et une évaluation rigoureuse garantissent la validité des résultats. Cependant, certaines limites sont à noter :

- **Généralisation des modèles** : Les performances des modèles Transformers restent influencées par la nature des données d'entraînement. L'article ne discute pas suffisamment l'impact potentiel des biais sur les résultats obtenus.
- **Dépendance aux données d'entraînement** : Les conclusions de l'étude pourraient ne pas être applicables à d'autres langages de programmation ou environnements, limitant ainsi leur portée.
- **Performance en conditions réelles** : L'intégration des modèles dans des environnements de développement (IDE) pourrait rencontrer des défis pratiques non abordés, tels que la latence du traitement et l'adoption par les développeurs.

Chapitre XII

Suggestions d'améliorations et directions futures

XII.1 Perspectives et améliorations possibles

L'article ouvre la voie à de nombreuses améliorations pour l'avenir de la détection de vulnérabilités en temps réel :

XII.1.1 Réduction des faux positifs

: L'un des principaux défis des outils automatisés est leur tendance à générer des alertes inutiles. La combinaison de méthodes basées sur des règles et des modèles d'apprentissage profond pourrait améliorer leur précision.

XII.1.2 Extension aux langages et frameworks variés

L'étude se concentre principalement sur quelques langages de programmation. Une généralisation à d'autres langages (Rust, Go, Swift) et frameworks (React, Angular, Spring) permettrait d'accroître l'utilité de ces outils.

XII.1.3 Étude de l'impact en entreprise

Pour évaluer réellement l'efficacité de ces modèles, une étude plus approfondie dans des contextes industriels est nécessaire. Cela permettrait de comprendre :

— L'acceptation par les équipes de développement.
— L'impact sur la productivité et la sécurité des applications.
— Les ajustements nécessaires pour un déploiement à grande échelle.

Conclusion

L'approche proposée par l'article constitue une avancée significative pour la détection automatique des vulnérabilités logicielles en temps réel. Toutefois, plusieurs défis restent à relever, notamment en ce qui concerne la généralisation des modèles, leur intégration dans des environnements de développement et leur adoption par les professionnels.

L'avenir de cette technologie dépendra de sa capacité à s'adapter à des contextes variés, à minimiser les biais et à proposer une approche fiable et efficace pour renforcer la sécurité des logiciels dès leur conception.

Cinquième partie

Bibliographie

Bibliographie

[1] *Awesome Automated Vulnerability Detection*. 2024. URL : https://github.com/alan-turing-institute/awesome-AVD.

[2] *Azure OpenAI Service deprecated models*.

[3] Aaron Chan et al. « Transformer-based Vulnerability Detection in Code at EditTime : Zero-shot, Few-shot, or Fine-tuning ? » In : (mai 2023).

[4] Mark Chen et al. « Evaluating Large Language Models Trained on Code ». In : (nov. 7). URL : https://arxiv.org/pdf/2107.03374.

[5] *GitHub Copilot : Your AI pair programmer*. Mars 2025. URL : https://github.com/features/copilot.

[6] Xinyi Hou et al. « Large Language Models for Software Engineering : A Systematic Literature Review ». In : (déc. 2024). URL : https://arxiv.org/pdf/2308.10620.

[7] *Machine Learning for Software Engineering*. URL : https://github.com/saltudelft/ml4se/blob/master/README.md.

[8] *OpenAI GPT-3 API : What is the difference between davinci and text-davinci-003*. 2023.

[9] Hammond Pearce et al. « Asleep at the Keyboard ? Assessing the Security of GitHub Copilot's Code Contributions ». In : (nov. 2020). URL : https://arxiv.org/pdf/2108.09293.

[10] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, nov. 2024. URL : https://udlbook.github.io/udlbook/.

[11] Chakraborty S., Pandey R. et Sinha S. « Deep Learning for Static and Dynamic Analysis of Code Vulnerabilities ». In : *In ACM Transactions on Software Engineering and Methodology* (2020).

[12] Alexey Svyatkovskiy et al. « IntelliCode Compose : Code Generation using Transformer ». In : (déc. 2021). URL : https://arxiv.org/pdf/2005.08025.

[13] *Tabnine : Industry-leading AI code assistant*. Mars 2025. URL : https://www.tabnine.com/about/.

[14] Microsoft Security Team. « AI-powered Code Security : A Comparative Analysis. » In : (2023). Microsoft Research Whitepaper.

[15] Towards an Understanding of Large Language Models in Software Engineering Tasks. « Zibin Zheng and Kaiwen Ning and Qingyuan Zhong and Jiachi Chen and Wenqing Chen and Lianghong Guo and Weicheng Wang and Yanlin Wang ». In : (déc. 2024). URL : https://arxiv.org/pdf/2308.11396.

[16] Ashish Vaswani et al. « Transformer-based Vulnerability Detection in Code at EditTime : Zero-shot, Few-shot, or Fine-tuning ? » In : *Advances in Neural Information Processing Systems* (2017).

[17] Hanxiang Xu et al. « Large Language Models for Cyber Security : A Systematic Literature Review ». In : (juill. 2024). URL : https://arxiv.org/pdf/2405.04760.

[18] Zibin Zheng et al. « A Survey of Large Language Models for Code : Evolution, Benchmarking, and Future Trends ». In : (jan. 2024). URL : https://arxiv.org/pdf/2311.10372.