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

31 mars 2025

Table des matières

Table des figures

Liste des tableaux

Première partie

Contexte et problématique

Les logiciels non sécurisés coûtent très cher aux entreprises, en dépit des moyens très avancés (outils, recherche) qui s'emploient à la recherche de vulnérabilités. En effet, une grande partie des vulnérabilités échappe aux outils de détection existants, laissant des failles potentielles de sécurité pour les utilisateurs et pour les systèmes d'information. Pour aggraver la situation, les délais entre la création effective de la vulnérabilité dans le code et sa découverte et correction effectives sont longs. La plupart des solutions d'analyse existantes exigent pour cela un code compilable et exécutable. Ce retard crée la même vulnérabilité mais à l'effet de la correction de plus grande complexité. Les millions de dépôts publics présents sur la plateforme GitHub traduisent cette montée en flèche des surfaces d'attaques, qui fait de la détection proactive une démarche incontournable pour déceler les vulnérabilités. Parallèlement, les modèles architecturaux pour toutes les applications devenant de plus en plus complexes, grâce aux frameworks, bibliothèques tierces et micro-services qui en multipliant les points d'injection potentielles, ouvrent de nouveaux vecteurs d'attaques. Dans un tel contexte, il apparaît clairement que le nouvel objectif est la détection des vulnérabilités dès l'EditTime du code, même s'il est partiel ni syntaxiquement erroné. Pour cela, l'objectif est d'offrir une détection interactive immédiate aux développeurs., afin qu'ils aient la possibilité de corriger les défauts de sécurité survenus dès leur apparition, sans attendre la phase de compilation/exécution. Ce concept s'inscrit parfaitement dans l'approche 'shift-left' en cybersécurité, inscrite dans le cœur du DevSecOps, qui vise à intégrer la sécurité le plus tôt possible dans le cycle de développement. Notons également qu'un bon nombre de vulnérabilités peut être imputé à la mauvaise connaissance ou la méconnaissance de la sécurité par les développeurs. Ainsi, en ayant une détection de failles dès le début du processus, il serait possible de réduire les risques et, par voie de conséquence, d'améliorer la sécurité des logiciels en général, grâce à une sensibilisation accrue.

Comment concevoir des outils capables de détecter de manière efficace et en temps réel les éventuelles vulnérabilités dans du code incomplet, afin à la fois de réduire les délais de correction, arrondir la qualité logicielle mais aussi de sécuriser le code dès les premières phases de son développement ?

Chapitre I

Présentation du domaine et des enjeux en cybersécurité

La cybersécurité prend une importance centrale alors que notre société s'automatise, la première vulnérabilité concernée est le logiciel, qui, comme tout produit technologique, souffre des défauts incriminés susceptibles de compromettre la sécurité des données, des utilisateurs et des systèmes. Pour faire face à ce défi majeur, deux techniques du logiciel de sécurité ont traditionnellement la faveur des experts, il s'agit de l'analyse dynamique ou de l'analyse statique. L'analyse dynamique est fondée sur l'exécution effective du code, révélant les vulnérabilités selon le comportement observé, mais souffre d'un fort problème de couverture persistant, rendant peu réalisables un trop grand nombre d'exécutions possibles du logiciel. L'analyse statique s'intéresse, elle, au code source ou le code binaire sans exécuter le logiciel, favorisant ainsi la couverture. Elle nécessite cependant la mise en place d'une définition manuelle des règles par des experts sur le terrain, ce qui rend compliqué un accompagnement en tant que stratégie de mesure de la sécurité, lourd d'efforts et donc ruineux eu égard à l'accélération de l'apparition de nouvelles menaces. En réponse à ces difficultés, l'intelligence artificielle, et singulièrement le machine learning et le deep learning, offre des voies prometteuses. En effet, les méthodes traditionnelles de machine learning reposent sur un ensemble de features extraites manuellement, provoquant une dépendance à l'humain et des charges lourdes au niveau de la maintenance des modèles. En revanche, les techniques de deep learning, et tout particulièrement celles basées sur des modèles de type transformers amènent à une automatisation des apprentissages des motifs de vulnérabilité directement à partir de grandes bases de données de code. Ces nouvelles méthodes constituent de réelles avancées en ce sens qu'elles permettent d'accroître très nettement la couverture et la précision des détections. Elles sont en outre bien plus adaptées à l'analyse de code auto-généré par les outils d'assistance au développement tels que GitHub Copilot ou Codex. Le raisonnement s'appuie sur l'hypothèse selon laquelle malgré leur indéniable efficacité, ces modèles d'IA générative peuvent introduire des vulnérabilités dans les logiciels, ce que rend d'autant plus problématique l'absence de contrôle. Une autre constante d'évolution est celle de la menace. Auparavant, elle pouvait se considérer comme technique, actuellement elle ne l'est plus, comme ces vulnérabilités exploitées massivement par des bots, ou ces attaques utilisant en toute discrétion des failles de type 0-day pour lesquelles aucun correctif n'existe sous forme de patch. La sécurité logicielle devient une exigence autant réglementaire, à travers des normes telles l'ISO/IEC 27001, la directive européenne NIS2, ou encore de la FDA pour les logiciels médicaux, qui avenant la nécessité d'être vigilant et du besoin de solidité des systèmes informatiques. De surcroît, la diversité et la multitude de vulnérabilités sont sources de complexité : le nombre de nouvelles vulnérabilités dans la base de données CVE/NVD dépasse le millier tous les ans, ce qui ne peut se gérer (et réagir) qu'avec des outils, humains, mais pas uniquement. D'où la nécessité accrue dans la stratégie des entreprises d'intégrer les modèles de langage (LLMs) en soutien au développement en intégrant la sécurité des solutions dès les premières étapes de développement, pour protéger efficacement les utilisateurs finaux et réduire de façon drastique le risque en production.

Chapitre II

Importance de la détection des vulnérabilités

Dès lors, énoncé de manière générale, la détection anticipative des vulnérabilités coutumières à se loger dans les logiciels paraît un souci essentiel, qui participe de la sécurité des programmes au bénéfice ultime de l'utilisateur final. Un point qui est précisé par de nombreuses études est que le temps de réaction suivant la vulnérabilité et la qualité de sa correction sont le plus souvent corrélés aux coûts de son exploitation par un hacker. D'après un rapport de l'IBM Security « Cost of a Data Breach 2023 » le coût d'une violation de données serait en moyenne de 4,45 millions de dollars actuellement, une valeur en constante hausse ! Il est vrai que plus une vulnérabilité est anciennement mise au jour, plus elle est difficile et donc plus coûteuse à combler. Une étude, parue en 2021 dans le Journal of Systems and Software semble le prouver, puisque dans le cas d'une vulnérabilité repérée au sein d'un logiciel dans sa phase de mise au point initiale par exemple, son coût peut être 30 fois plus faible en cas d'intervention qu'à l'issue du processus de production phase au cours de laquelle elle a pu être exposée à l'exploitation en ligne active par les cybercriminels. Le coût ici correspond aux coûts techniques du remplacement de code informatique mais aussi aux pertes économiques indirectes à l'exploitation de la vulnérabilité. Il existe une multitude de vulnérabilités critiques, telles les injections SQL, les Cross-Site Scripting (XSS), et les erreurs dans la validation d'un code qui se révèlent dès les premières écritures du code. Selon une étude menée par OWASP (Open Web Application Security Project) plus de 70% des vulnérabilités référencées au sein du Top 10 de OWASP 2021 auraient pu être repérées et corrigées dès la saisie de code (EditTime). La détection en ligne de leurs vulnérabilités permet ainsi d'en corriger les failles , et in fine de réduire fortement, les coûts d'intervention ultérieurs, tout en bénéficiant de l'amélioration substantielle de la qualité du code produit. En plus d'épargner ces coûts, une telle anticipation devient précieuse pour réduire la fenêtre d'exposition, aux cyberattaques. Actuellement, d'après le rapport de Veracode "State of Software Security 2023" , le temps moyen allant de l'introduction d'une vulnérabilité à sa prise en compte dépasse généralement les 200 jours. Les applications sont alors longuement laissées, exposées, aux menaces numériques tiers, malveillantes. En réagissant à la découverte du défaut au moment même où celui s'introduit, la menace étant prise en compte dans les heures voire les en temps réel suivant, la protection se renforce pour les données sensibles, et une continuité des services est préservée. Les effets d'une exploitation de faille en production sont lourdement dommageables, exposant la fuite de données sensibles, le choc d'une réputation sévèrement atteinte, des amendes parfois très éprouvantes comme celles de l'application du RGPD (jusqu'à 4chiffre d'affaires annuel consolidé d'une entreprise) et des pertes d'activité en conséquences parfois existentielles. Par ailleurs, selon l'étude de Gartner (2022), 60entreprises sont déclarées disparues dans les six mois suivant une cyberattaque majeure du fait des coûts générés pourtant directs en premier lieu comme indirects en second lieu. En outre, enfin, en adoptant une stratégie de détection des vulnérabilités en EditTime, on peut avoir l'ambition d'une vraie valeur pédagogique, ayant en interaction directe et immédiate avec les bonnes pratiques et erreurs potentielles des développeurs et prenant ainsi une disponibilité sans cesse accrue de la conscience et du niveau leurs compétences en sécurité applicative. En définitive cela promeut une culture pro-active et résistante de sécurité logicielle au sein des équipes, par là même réduisant cumulativement la dette technique, suffisant ainsi à consolider la résilience à long terme des systèmes d'information.

Chapitre III

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

Les méthodes classiques de détection des vulnérabilités souffrent de plusieurs limites essentielles, qui nuisent à leur efficacité dans un mode de détection instantané. Premièrement, ces méthodes classiques sont, notamment les outils d'analyse statique, comme CodeQL, nécessitant quasiment un code source complet et sans erreur syntaxique. Dans l'article « Evaluating Static Analysis Tools for Vulnerability Detection » IEEE Transactions on Software Engineering (2022), il est montré que 40non détectées pour un code source incomplet ou comportant des erreurs minimes. De ce fait, il est évident que ce type de méthode s'avère peu pertinent pour une détection instantanée dès la phase d'écriture du code. Qui plus est, ces méthodes reposent sur des règles rigides et existentiellement définies, dont le maintient à jour nécessite en permanence de gros efforts humains. Ainsi une étude du Journal of Information Security and Applications (2021) estime que 25 seul travail, ce qui se traduit par un coût humain difficilement acceptable. Dès lors, à chaque émergence de nouvelles vulnérabilités critiques dans une application correspond des efforts considérables pour adapter au mieux les règles statutaires existantes, limitant ainsi leur capacité d'adaptation face à de nouveaux types de vulnérabilités. Selon l'approche adoptée (analyse statique ou dynamique), ces outils classiques sont souvent sujets à des compromis sévères, comme le montre une étude du NIST de 2021 qui a révélé que les outils statiques génèrent jusqu'à 30 confiance des développeurs, quand les vulnérabilités critiques leur échappent souvent dans les outils dynamiques, avec environ 20 de la détection de vulnérabilités en temps réel (EditTime) s'additionne donc à d'autres défis spécifiques. L'un des verrous principaux réside dans l'analyse efficace de fragments de code souvent incomplets ou syntactiquement erronés en cours de saisie, qui nécessite le recours à des modèles analytiques à faible latence, très précision et aisément intégrables aux environnements de développement intégrés (IDE) tels que VSCode ou IntelliJ, sans compromettre la productivité du développeur. Le compromis délicat entre précision et rappel doit également être respecté : si l'on veut donner confiance aux développeurs, il faut réduire le nombre de faux positifs, donc maximiser la couverture des vulnérabilités majeures. Comme le montre une étude de l'Empirical Software Engineering Journal (2023), au-delà de 10 équipes techniques (tutoiement utilisé dans la suite) finissent souvent par arrêter d'utiliser l'outil. La scalabilité est aussi un défi important, les outils devant pouvoir traiter efficacement différents langages de programmation, différents contextes applicatifs et diverses vulnérabilités. La gestion adéquate du code produit « automatiquement » par des modèles de type LLM (Large Language Models) complexifie encore le tout, et notamment à cause de la multiformité et la structure imprévisible de ce type de code. De plus, certaines vulnérabilités complexes, comme les injections SQL avancées ou les erreurs d'authentification plus classiques, nécessitent une bonne connaissance du flux des données et un bon repérage du contexte général de l'application complexifiant ainsi l'analyse instantanée et pertinente. Enfin, pour être largement adoptés par les développeurs, ces outils doivent être simples, voire légers, conviviaux et facilement personnalisables selon le besoin de chaque équipe ou de chaque entreprise. Un déploiement fluide et adaptable dans les IDE modernes peut être un critère important pour une adoption efficace et pérenne des solutions de détection des vulnérabilités en temps réel.

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 – Performances des modèles étudiés

Troisième partie

Impacts et applications

Chapitre VII

Améliorations du développement logiciel

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

Comme vu en II, l'état de l'art en détection de vulnérabilités logicielles intègre déjà l'utilisation de LLM en aval de la phase de développement. Cette section propose une courte rétrospective des méthodes de détection de vulnérabilités classiques afin de mieux cerner la plus-value apportée par la détection en temps réel.

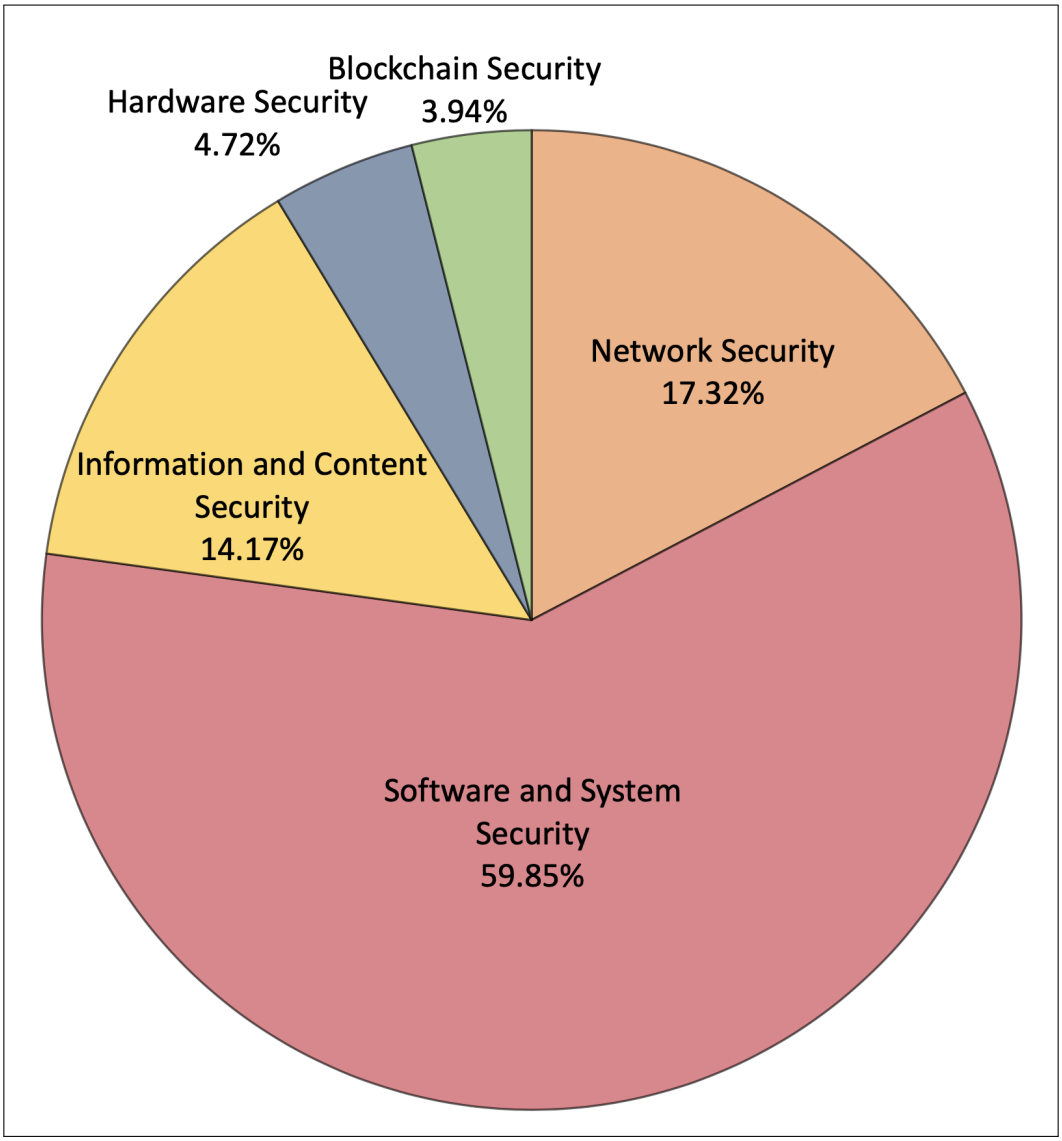

FIGURE VII.1 – Répartition de l'utilisation de LLM pour des tâches de sécurité[22]

VII.1.1 Détection de vulnérabilités par IA : quelques outils à l'état de l'art

VII.1.1.1 Copilot

`Copilot` peut facilement être considéré comme le leader du développement assisté par LLM. Ses fonctionnalités sont nombreuses, bien éprouvées, et portées sur plusieurs modèles de pointe *(GPT-4o, Claude 3.7 Sonnet ...)*. `Copilot` est désormais intégré sur de nombreux environnements de développement *(VSCode, Suite Jetbrains ...)* et utilisé de façon quasi universelle.

Par ailleurs, `Copilot` a servi de base de développement à un outil sorti en 2024 visant spécifiquement à introduire l'IA dans la cybersécurité : `Microsoft Copilot for Security` [1]. Si l'entreprise vante les mérites d'une analyse évolutive en temps réel de l'outil [2], la présente étude n'a pas trouvé de source mentionnant une capacité de celui-ci à détecter des vulnérabilités logicielles sur cette même temporalité. Cette feature est donc vraisemblablement absente ou méconnue des utilisateurs.

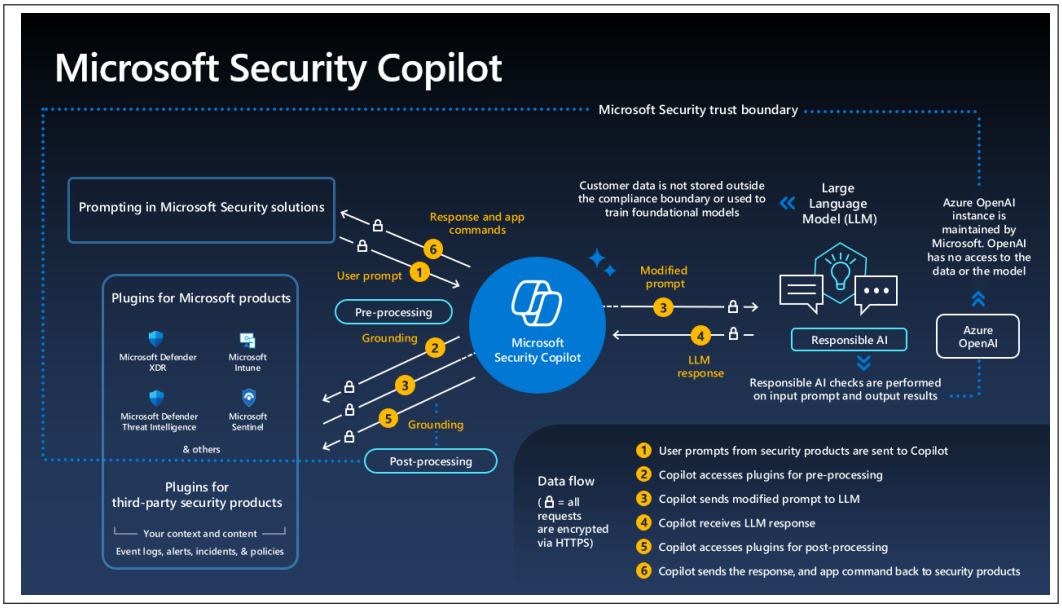

FIGURE VII.2 – Schéma conceptuel du fonctionnement de `Microsoft Copilot for Security`

VII.1.1.2 `Snyk`, *powered by* `Deepcode` : un exemple de standard industriel pour la sécurisation de code par IA

Cet outil permet de mener des analyses statiques de code à des fins spécifiques de sécurisation. Cependant, du propre aveu du développeur [3], l'outil ne propose pas systématiquement de correctif par peur de remplacer une vulnérabilité par une autre, ou de ne pas arriver à corriger celle identifiée.

Par ailleurs, les analyses statiques ne permettent par définition pas à`Snyk` ne permet pas de détecter les vulnérabilités en temps réel. Il est donc impossible d'utiliser cet outil pour corriger le code pendant la phase de développement.

VII.1.2 Cas particulier du code généré par des LLM

Dans une perspective de nouveau paradigme de développement logiciel, il est important de considérer que le code revu par LLM peut également être un code généré par de tels modèles.

Pratiques actuelles de développement par LLM : exemple d'IntelliCode Considéré comme le premier LLM développé pour générer du code [24], `IntelliCode` est un outil moins récent que les deux précédemment cités *(sa première publication académique datant de 2020[18])*, mais encore largement utilisé, particulièrement dans la suite `Visual Studio`.

La présente étude n'a pas trouvé de référence faisant explicitement part des aspects orientés sécurité de l'outil. Cette absence peut s'expliquer par la volonté de Microsoft de capitaliser sur les fonctionnalités plus récentes de `Microsoft Copilot for Security` pour en faire à terme un standard et laisser `IntelliCode` à son stade plus expérimentale.

1. https ://www.youtube.com/watch ?v=T3OKmtIPyzQ
2. https ://www.lemagit.fr/actualites/366621612/Microsoft-pousse-Security-Copilot-dans-lere-agentique
3. https ://docs.snyk.io/scan-with-snyk/snyk-code/manage-code-vulnerabilities/fix-code-vulnerabilities-automatically

VII.2 Interprétation des métriques de classification présentées

Les modèles présentés sont évalués selon les méthodes de Pearce et. Al [13], c'est-à-dire en se basant sur les scénarios prévus par le classement `MITRE` des 25 vulnérabilités les plus importantes. En outre, ces scénarios prévoient une prévalence de certaines vulnérabilités *(les injections SQL par exemple)*.

Comme vu dans les parties précédentes, les métriques d'évaluation à proprement parler sont celles propres à un problème de classification, puisqu'il s'agit en l'occurrence d'une classification binaire *(code vulnérable ou code non vulnérable)*. Les trois critères retenues sont la précision, le rappel et le score F1. Leurs expressions sont rappelées ci-dessous :

$$\text{precision} = \frac{TP}{TP + FP} \quad \text{rappel} = \frac{TP}{TP + FN} \quad \text{score F1} = 2 \times \frac{\text{precision} \times \text{rappel}}{\text{precision} + \text{rappel}} \tag{VII.1}$$

Si ces métriques sont universelles et facilement interprétables, celles-ci sont parfois insuffisantes pour évaluer finement la performance d'un modèle de classification. C'est d'ailleurs ce qui est rappelé dans l'article cité dans la partie sur les métriques d'évaluation [**40article**]. Celui-ci propose de compléter les trois critères cités précédemment par des métriques moins triviales et moins biaisées au sens de l'auteur.

VII.3 Correction et complétion pendant la phase de développement : promesses et difficultés rencontrées

Les parties précédentes montrent que la détection de vulnérabilités en temps réel est grandement améliorée par les LLM développés, en particulier `CodeBERT` fine-tuné. Cette conclusion est valable pour le code généré par IA comme pour le code "strictement humain".

À la lumière des sections de ce chapitre, on peut affirmer que ces résultats sont prometteurs pour le secteur du développement logiciel, la correction par LLM lors de la phase de développement n'ayant été proposée par aucun outil à l'état de l'art.

On relève cependant plusieurs points d'amélioration notables : aucun outil de développement par LLM n'est actuellement sensiblement meilleur que ses concurrents *(sur les aspects de sécurité mais pas exclusivement)*, ne laissant pas la possibilité à court de terme de voir la naissance d'un nouveau standard. Par ailleurs, les LLM présentés ici nécessitent vraisemblablement plusieurs autres phases d'évaluation plus poussées avant un déploiement industriel. Ces remarques serviront de point de départ pour le prochain chapitre.

Chapitre VIII

Conséquences pour l'industrie et la recherche

Nous avons entrevu enI les coûts réels des vulnérabilités logicielles pour les entreprises et les institutions publiques. La littérature scientifique à ce sujet est vaste[2],[1] et montre l'importance que revêt le développement et le déploiement d'outils de détection de vulnérabilités logicielles plus performants.

VIII.1 Projets industriels

Malgré quelques recherches orientées vers des projets en cours menés par des entreprises, et pour des raisons évidentes de confidentialité, il est difficile d'estimer comment les résultats de l'article peuvent s'inscrire à court terme dans l'industrie du logiciel. On peut tout de même citer les travaux de NVIDIA visant à moderniser les outils d'énumération traditionnels [1] ainsi que d'autres articles proposés par des structures externes au monde de la recherche [2], [3].

VIII.2 Continuité de la recherche

Un peu moins de deux ans après la publication originale de l'article, on note d'une part la démocratisation du développement logiciel assisté par LLM [23] et d'autre part les problèmes de sécurité propres à ces nouvelles méthodes de développement[10, 25, 24, 22].

VIII.2.1 Développement par LLM et évolution en conséquence des méthodes de détection de vulnérabilité

Les publications citées *(s'appuyant toutes sur l'article principal)* ne présentent pas de consensus quant à la différence de niveau de sécurité entre le code partiellement ou totalement généré, et le code développé par les seules méthodes conventionnelles. Cette absence de consensus est notamment due à l'apparition très récente des LLM dans les projets industriels et académiques.

En revanche, on peut esquisser deux conclusions :

— Si le niveau de vulnérabilité ne diminue pas de manière universelle avec l'emploi de LLM, il est tout de même crucial de revoir les méthodes de détection traditionnelles car les vulnérabilités générées par LLM ne sont pas nécessairement semblables à celles générées par le développement strictement humain

— En particulier, il apparaît que le développement par LLM peut prendre des formes plurielles et renforce paradoxalement l'importance du facteur humain pendant les phases de développement. À cet égard, il serait intéressant de définir plus finement ce que l'on nomme "développement par LLM" et selon une métrique plus normative *(taille des prompts, part des éléments du projet "délégués" à l'IA, profil et qualifications des développeurs ...)* pour préciser les résultats des publications étudiées

1. https://docs.nvidia.com/nemo/guardrails/latest/evaluation/llm-vulnerability-scanning.html
2. https://www.tigera.io/learn/guides/llm-security/
3. https://www.securityjourney.com/ai/llm-tools-secure-coding

VIII.2.2 Large Language Models et détection de vulnérabilités : promesses et limites

VIII.2.2.1 Sélection et préparation des données : des méthodes encourageantes

L'article principal a ouvert la voie à un nouveau paradigme de programmation reposant largement sur l'utilisation de LLM d'une part pour la génération de code et d'autre part pour la détection de vulnérabilités dans ce même code.

En particulier, les méthodes de collecte et de préparation des données d'entraînement sont saluées et reprises dans les publications récentes citées plus haut.

Expliquer en détail ces méthodes dépasserait le cadre de cette étude, mais on l'on peut souligner que celles-ci reposent sur la définition d'une corrélation entre les caractéristiques des données, les tâches de sécurité et le comportement des LLMs.

VIII.2.2.2 Failles du modèle et pistes d'amélioration

Compromis entre taille du modèle et délai de réponse L'article et les publications associées insistent sur l'importance de la rapidité du modèle, par définition. En effet, la contrainte principale étant d'identifier les vulnérabilités en temps réel, celui-ci doit présenter un temps de réflexion maximum inférieur aux valeurs que l'on peut observer lors d'une utilisation "classique" de LLM comme Claude Sonnet 3.7 ou GPT-4o.

L'enjeu est de définir un modèle suffisamment rapide pour que le développeur soit averti de son erreur avant de passer à l'étape suivante de son projet. À ce stade, les publications étudiées font peu mention de ce problème, et encore moins des moyens pour le résoudre.

Vulnérabilités non détectées Les vulnérabilités couvertes par les données d'entraînement sont rappelées par le tableau suivant :

Vulnerability CWE	N	
SQL Injection	89	45
Hardcoded Credentials	798	23
Code Injection	94	13
Path Injection	22	7
Clear Text Logging	312	5
Weak Cryptographic Algorithm	327	5
Incomplete URL Substring Sanitization	20	2

TABLE VIII.1 – Statistiques récapitulatives des problèmes de vulnérabilité recueillis à partir des PR GitHub[5]

Le choix de se concentrer sur ces vulnérabilités n'est pas anodin et découle d'une réflexion nourrie par le framework `MITRE`. Cependant, plusieurs vulnérabilités potentiellement critiques ne sont pas couvertes. Citons en particulier des vulnérabilités pour le développement Web de type :

— Cross-Site Scripting (XSS)
— Cross-Site Request Forgery (CSRF)
— XML External Entity (XXE)
— Fuzz Testing
— DDoS

Ces vulnérabilités peuvent entraîner l'exfiltration de données sensibles, voire la prise de contrôle de sessions utilisateurs.

Par ailleurs, notre étude rappelle que les différents types de vulnérabilités ont été étudiées séparément [5] [13], le framework d'évaluation visant initialement à identifier quelles vulnérabilités pouvaient être générées par `Copilot` et sous quelles conditions. Cependant, dans le cadre de l'article principal, il aurait été intéressant de confronter les modèles à un projet se rapprochant plus de la réalité, où différentes vulnérabilités peuvent coexister.

Langages non couverts L'étude a choisi de se concentrer sur 7 langages de programmation universels. Cette sélection est difficilement contestable si l'on se réfère à leur omniprésence historique et actuelle dans le développement logiciel *(Indice TIOBE en mars 2025*[4]*)*.

4. `https://www.tiobe.com/tiobe-index/`

Cependant, on peut penser que l'étude pourrait renforcer son impact si elle était prolongée sur des langages moins utilisés mais tout de même bien ancrés dans le paysage du développement *(on pense immédiatement à Rust pour le développement logiciel, et on pourrait citer R ou Julia pour étendre la portabilité de l'outil sur les logiciels de calcul scientfiques)*.

Langage	**Nombre de CWE couvertes**	**Codes vulnérables**	**Codes non vulnérables**
`Javascript`	70	266,342	2,293,712
`Python`	37	149,158	1,493,972
`Go`	29	50,233	535,180
`Java`	44	33,485	431,726
`C++`	32	7,222	215,722
`C#`	54	3,341	27,731
`Ruby`	19	137	1,957

TABLE VIII.2 – Quelques statistiques sur les données d'entraînement[5]

Le double défi de la confidentialité et de la transparence À ce stade, les autres problématiques soulevées par l'utilisation de LLM n'ont pas été abordées. Si elles dépassent le strict cadre de l'étude bibliographique, il n'est pas inutile de rappeler que la sécurisation des modèles devra s'accompagner d'une prise en compte de ces problématiques.

Ainsi, les modèles présentés dans l'article n'échappent pas à ces problèmes de transparence et de confidentialité, bien que ce dernier n'en fasse pas mention. On peut citer [9] *(publié notamment par l'INSA Rennes)* comme proposition d'un LLM de détection de vulnérabilités faisant mention explicite de ses considération en matière de transparence, notamment eu égard à la collecte et au traitement des données d'entraînement. Les deux articles proposant un modèle fonctionnant sous `BERT` [5], leur étude croisée pourrait aboutir à une approche plus globale quant au développement logiciel par LLM.

5. on rappelle que le modèle `CodeBERT` est le plus efficace des trois présentés plus haut

Quatrième partie

Analyse critique et perspectives

Chapitre IX

Problèmes éthiques et limites des modèles d'IA

IX.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 :

IX.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.

IX.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.

IX.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 X

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 XI

Suggestions d'améliorations et directions futures

XI.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 :

XI.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.

XI.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.

XI.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] Afsah Anwar et al. « Measuring the Cost of Software Vulnerabilities ». In : (mai 2020). URL : https://d1wqtxts1xzle7.cloudfront.net/94423818/eai.13-7-2018-libre.pdf?1668727337=&response-content-disposition=inline%3B+filename%3DMeasuring_the_Cost_of_Software_Vulnerabi.pdf&Expires=1743259891&Signature=OXN2-NDnTwqgEuyxIaXR4DRTA3jChwv624CgaMt6cM6enpQxXbQbtqBUY9yXNQ2waViwCtR5oWMMslp~57gcysxNIzL4Tf6m~ODTtOBY7f2nZIhdlMYnD4vnQZ-QaKBnYp5IyU6FvnR~mcHNPrgGcyOuj~vmtEhtP9c23go_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

[2] Afsah Anwar et al. « Understanding the Hidden Cost of Software Vulnerabilities : Measurements and Predictions ». In : (2018). URL : https://www.cs.ucf.edu/~mohaisen/doc/sc18.pdf.

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

[4] *Azure OpenAI Service deprecated models*.

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

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

[7] *Copilot amplifies insecure codebases by replicating vulnerabilities in your projects*. Fév. 2024. URL : https://snyk.io/blog/copilot-amplifies-insecure-codebases-by-replicating-vulnerabilities/.

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

[9] Jean Haurogné, Nihala Basheer et Shareeful Islam. « Vulnerability detection using BERT based LLM model with transparency obligation practice towards trustworthy AI ». In : (nov. 7). URL : https://www.sciencedirect.com/science/article/pii/S2666827024000744.

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

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

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

[13] 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.

[14] David Powers. « Evaluation : From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation ». In : (2020). URL : https://arxiv.org/pdf/2010.16061.

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

[16] 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).

[17] Shaznin Sultana, Sadia Afreen et Nasir U. Eisty. « Code Vulnerability Detection : A Comparative Analysis of Emerging Large Language Models ». In : *Boise State University* (2024). URL : https://arxiv.org/pdf/2409.10490.

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

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

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

[21] Ashish Vaswani et al. « Attention is all you need ». In : *Advances in Neural Information Processing Systems* (2017).

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

[23] Ziyin Zhang et al. « Unifying the Perspectives of NLP and Software Engineering : A Survey on Language Models for Code ». In : *Preprint* (juin 2025). URL : https://arxiv.org/pdf/2311.07989.

[24] 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.

[25] Zibin Zheng et al. « Towards an Understanding of Large Language Models in Software Engineering Tasks ». In : (déc. 2024). URL : https://arxiv.org/pdf/2308.11396.